AFIT/GCS/ENG/91D-22

AN APPLICATION OF THE OBJECT-ORIENTED PARADIGM TO A FLIGHT SIMULATOR

THESIS

Dennis Joseph Simpson
Captain, USAF

AFIT/GCS/ENG/91D-22

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
AN APPLICATION OF THE OBJECT-ORIENTED PARADIGM TO A FLIGHT SIMULATOR

5. FUNDING NUMBERS

6. AUTHOR(S)
Dennis J. Simpson, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-22

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
RL/COAA
Griffis AFB, NY 13441

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This thesis describes the object-oriented software development techniques that were used to analyze, design and implement a flight simulator. The objective of this thesis was to present a comprehensive object-oriented software development methodology and show how it was used in constructing an actual application.

An extensive review of current object-oriented practices is presented along with the methods that were used to take the flight simulator from analysis to design and through to implementation. The description of the methodology concentrates upon the design and implementation phases of the object-oriented software lifecycle. Examples from the design of the flight simulator demonstrate how each phase of the methodology was applied. The thesis includes insights on how to use the C++ language in implementing an object-oriented design and how to fold procedurally oriented code into an object-oriented framework.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer Applications, Computer Graphics, Computer Programming, Flight Simulation, Software Engineering	191
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave Blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency. Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ..., To be published in .... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availablity Statement. Denote public availability or limitation. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR)

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA	- NASA - Leave blank
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

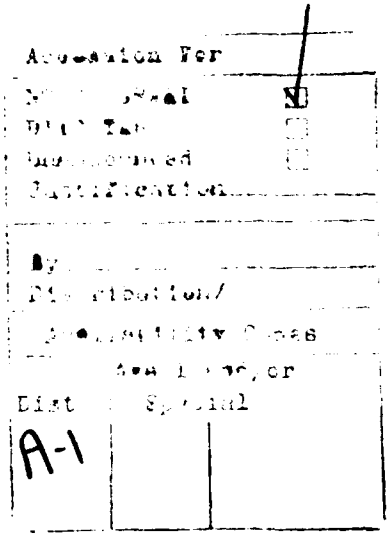
Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

AFIT/GCS/ENG/91D-22

AN APPLICATION OF THE OBJECT-ORIENTED PARADIGM TO A FLIGHT SIMULATOR

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)



Dennis Joseph Simpson, B.S.
Captain, USAF

December, 1991

Acknowledgments

I am indebted to many individuals. I would like to thank my advisor, Lieutenant Colonel Phil Amburn, for his help and guidance during this project. I also want to thank the other members of the thesis group: Captain John Brunderman, Captain Bob Olson, Captain Mark Gerken, and Captain Don Duckett. This would not have been possible without them. Additional thanks goes to my readers, Lieutenant Colonel Pat Lawlis and Lieutenant Colonel Marty Stytz and to Major David Umphress for his "object-oriented" advice way back when this all started.

My greatest thanks goes to my wife De. Thanks for hanging in there through those lonely times and not letting me open those computer games until this was finished.

Dennis Joseph Simpson

Table of Contents

		Page
Acknowledgments		ii
Table of Contents		iii
List of Figures		vii
List of Tables		ix
Abstract		x
I.	Introduction	1-1
	1.1 Background	1-1
	1.2 Problem	1-1
	1.3 Scope	1-2
	1.4 Assumptions	1-2
	1.4.1 Implementation	1-2
	1.5 Summary	1-4
	1.6 Thesis Overview	1-4
II.	Literature Review	2-1
	2.1 Introduction	2-1
	2.2 The Software Crisis	2-1
	2.3 The Object-Oriented Paradigm	2-2
	2.3.1 Objects	2-3
	2.3.2 Classes	2-3
	2.3.3 Relationships Between Classes	2-3

Page

2.3.4 Communication Between Objects 2-5

2.3.5 Terminology 2-5

2.4 Major Principles of the Object Oriented Paradigm . . . 2-7

2.4.1 Abstraction 2-7

2.4.2 Encapsulation 2-8

2.5 How To Apply the Object-Oriented Paradigm 2-8

2.5.1 Object-Oriented Analysis (OOA) 2-9

2.5.2 Object-Oriented Design (OOD) 2-16

2.5.3 Object-Oriented Programming (OOP) 2-27

2.5.4 Notation 2-28

2.6 Benefits and Drawbacks of the Object-Oriented Approach 2-28

2.6.1 Benefits of the Object-Oriented Approach . . 2-28

2.6.2 Drawbacks of the Object-Oriented Approach . 2-37

2.7 Successful Applications of the Object-Oriented Approach 2-39

2.8 Summary 2-40

III. Methodology 3-1

3.1 Introduction 3-1

3.2 Analysis 3-3

3.2.1 The Analysis Process 3-4

3.2.2 The Notation 3-8

3.3 Design 3-11

3.3.1 Good Design/Designing For Reusability 3-12

3.3.2 High Level Design 3-14

3.3.3 Low Level Design/Implementation 3-21

3.4 Conclusion 3-25

Page

IV. Design Highlights 4-1

4.1 Introduction 4-1

4.2 Analysis 4-1

4.2.1 Initial Analysis 4-2

4.3 Low Level Inputs — The Joystick and RS232 Port Classes 4-6

4.3.1 Detailed Analysis 4-7

4.3.2 High Level Design I 4-9

4.3.3 Low Level Design/Implementation I 4-12

4.3.4 High Level Design II 4-17

4.3.5 Low Level Design II — Revising the Joystick Class 4-20

4.3.6 Final Design Activities 4-22

4.4 Static Data Members and Static Methods 4-23

4.5 The Window and Text Window Classes 4-24

4.5.1 Resolving Multiple Relationships 4-25

4.6 Conclusion 4-29

V. Reusing Procedurally Oriented Code 5-1

5.1 Introduction 5-1

5.2 Reasons for Folding Procedures into Classes 5-1

5.3 Identify the Candidate Class 5-3

5.4 Identifying the Data and Procedures 5-4

5.5 Implementing the Methods 5-5

5.6 Conclusion 5-8

VI. Summary and Recommendations 6-1

6.1 Summary 6-1

6.1.1 Literature Review 6-1

Page

6.1.2 Methodology 6-5

6.1.3 Applying the Methodology 6-9

6.1.4 Reusing Procedurally Oriented Code 6-9

6.2 Conclusions 6-10

6.3 Recommendations for Future Research 6-10

6.4 Remarks 6-11

Appendix A. Bibliography of OOA Sources A-1

Appendix B. Flight Simulator Context Analysis B-1

Appendix C. Flight Simulator Design C-1

Appendix D. Bibliography of Additional Object-Oriented Sources . D-1

Appendix E. Using C++ To Implement an Object-Oriented Design E-1

E.1 Introduction E-1

E.2 Object Representation E-2

E.3 Providing Object Visibility E-4

E.3.1 Implementing Required Data Paths E-5

E.3.2 Hiding Data E-8

E.4 Conclusion E-13

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure — Page

2.1. The Object-Oriented Structured Design Notation 2-29
2.2. Coad and Yourdon's OOA/OOD Notation 2-30
2.3. Booch Class Diagram 2-31
2.4. Booch Object Diagram 2-32
2.5. Booch Module and Process Diagrams 2-33

3.1. System Notation 3-9
3.2. Object Notation 3-10

4.1. Flight Simulator Initial Analysis 4-5
4.2. Initial Joystick Attributes 4-8
4.3. Initial Joystick Methods 4-9
4.4. Initial Implementation of the Joystick and RS232 Port Classes . . 4-16
4.5. Second Design of Joystick and RS232 Port Classes 4-18
4.6. Full Design of the Joystick and RS232 Port Classes 4-21
4.7. The Window Class Heirarchy 4-26

B.1. Flight Simulator Concept Map B-1

C.1. Flight Simulator Composition C-1
C.2. Window and Queued Input Classes C-2
C.3. Dynamic Object Class Hierarchy C-3
C.4. Input Devices C-4
C.5. RS232 Port Class Hierarchy C-5
C.6. World Window Class Composition C-6
C.7. Image Generation Classes C-7

Figure Page

C.8. Distributed RS232 Port Class C-8

C.9. Dynamic Objects Manager C-9

C.10. Distport Module C-10

C.11. Dynamic Object Class C-11

C.12. F16 Class C-12

C.13. Flight Simulator Class C-13

C.14. Font Class C-14

C.15. Font Manager C-15

C.16. Force Torque Spaceball Class C-16

C.17. Joystick Class C-17

C.18. Managed RS232 Port Class C-18

C.19. RS232 Port Class C-19

C.20. Port Manager C-20

C.21. Port Reader Class C-21

C.22. Queued Inputs Manager C-22

C.23. Queued Input Class C-23

C.24. Socket Class C-24

C.25. Spaceball Class C-25

C.26. Spaceball Port Reader Class C-26

C.27. Static Timer C-27

C.28. Text Item Class C-28

C.29. Text Window Class C-29

C.30. Unmanaged RS232 Port Class C-30

C.31. User Aircraft Class C-31

C.32. Voltages Spaceball Class C-32

C.33. Window Manager C-33

C.34. Window Class C-34

List of Tables

Table		Page
2.1.	Comparison of OOA Techniques	2-11

Abstract

This thesis describes the object-oriented software development techniques that were used to analyze, design and implement a flight simulator. The objective of this thesis was to present a comprehensive object-oriented software development methodology and show how it was used in constructing an actual application.

An extensive review of current object-oriented practices is presented along with the methods that were used to take the flight simulator from analysis to design and through to implementation. The description of the methodology concentrates upon the design and implementation phases of the object-oriented software lifecycle. Examples from the design of the flight simulator demonstrate how each phase of the methodology was applied.

The thesis includes insights on how to use the C++ language in implementing an object-oriented design and how to fold procedurally oriented code into an object-oriented framework.

AN APPLICATION OF THE OBJECT-ORIENTED PARADIGM TO A FLIGHT SIMULATOR

I. Introduction

1.1 Background

"The object of flight simulation is to reproduce on the ground the behavior of an aircraft in flight" (45:1). The benefits of flight simulators include saving money, increased safety, more opportunity for use (versus flying the actual aircraft) and less harm to the environment (45:234-235). One of the disadvantages of flight simulators is that they can cost millions of dollars (63:20).

The majority of flight simulators in use today are made where the "out the cockpit" images are entirely computer generated. In effect, the computer places the pilot completely in an artificial world of the computer's creation. The main benefit of this "virtual environment" is that what the person can see or do is limited only by the imagination of the computer programmer.

Thesis students at the Air Force Institute of Technology (AFIT) have been researching various aspects of low cost (under $100,000) flight simulators and virtual environment systems since 1988. These research efforts have produced a variety of software and hardware tools that can be reused in future applications (17, 40, 43). The overall goal of this research at AFIT is to investigate the applicability and value of such systems to the operational Air Force.

1.2 Problem

The focus of this thesis is the software design and implementation of a flight simulator. The design of the flight simulator must serve as the foundation for current

and future AFIT research into flight simulation and virtual worlds. The design of the new flight simulator must promote the following principles:

Modifiability If the design is to serve as a foundation for research, then it must be easy to change.

Extensibility Winblad defines extensibility as "the ability of a program or system to be easily altered" (60:265). The design must be constructed in such a manner that future additions to the design can be made with a minimum of effort. *Adding* to the design should not mean *redesigning* it from scratch.

Reusability The design should provide a format for reusing existing code and a framework that enables the reuse of new code.

1.3 Scope

The subject of this thesis is the design and implementation of a flight simulator using an object-oriented methodology. The design was implemented using the C++ programming language. The two main objectives of this thesis are to present a comprehensive, object-oriented software development process and to show how this process was applied to the implementation of the flight simulator.

1.4 Assumptions

1.4.1 Implementation The flight simulator was a group effort. There were four other individuals that worked on closely related projects. Because the topics were so closely related, we were able to reuse code and parts of the flight simulator design between projects. These projects are discussed in the theses by Brunderman, Duckett, Gerken and Olson (8, 14, 20, 38).

1.4.1.1 Software We had access to the following sources of software:

1. The C++ Programming Language: "Not surprisingly, the most natural implementation target for an object-oriented design is an object-oriented language" (47:296). "A language *supports* a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style" (56:19). The C++ programming language supports the object-oriented programming style (56). Both the the AT&T C++ Translator and GNU C++ provided by the Free Software Foundation were available for our use.

   The decision to use the C++ language was strictly pragmatic. We did not have access to another object-oriented language that would run on the platforms upon which the design was implemented.

2. Silicon Graphics and 4Sight Window Manager Libraries: Further software support is provided by the library of routines that comes with the Silicon Graphics machines. These library routines allow the programmer to control the graphics operations of the machine, keyboard input and the 4Sight window system.

3. UNIX Standard Library: We were working within the UNIX operating system. We therefore were able to use UNIX standard system calls and "C" software libraries.

4. Previous Thesis Efforts: The last source of code was code written in previous thesis efforts. The routines that Captain Bob Filer wrote for controlling the input devices used in his virtual environment display system were reused in the flight simulator (although not in their pristine condition) (17). Captain Phil Platt's code was used to help determine how to interface the joysticks with the flight dynamics (40).

1.5 Summary

AFIT has been involved in research into low cost flight simulators and virtual worlds since 1988. This year's efforts built upon past efforts and established a framework for future endeavors. This framework had to be modifiable, extendable and had to facilitate reuse. This framework for future research took the form of an object-oriented design for a new flight simulator.

The objective of this thesis is to present an object-oriented software development process that can be used to take an application from problem definition through to implementation. These concepts will be discussed with respect to the actual design and implementation of the flight simulator.

1.6 Thesis Overview

This document contains 6 chapters. Chapter 2 is a literature review of the *object-oriented paradigm*. Chapter 3 explains the methodology used in the design of the flight simulator. Chapter 4 highlights specific aspects of the methodology with examples from the flight simulator. Chapter 5 describes how to reuse procedurally oriented code in an object-oriented design. Chapter 6 reports on the results of the thesis, gives suggestions for future research and provides some final comments.

II. Literature Review

2.1 Introduction

The purpose of this chapter is to lay the foundation for the discussions contained in the following chapters. This chapter begins with an explanation of the motivation for the object-oriented paradigm — the software crisis. This explanation is followed by a description of the object-oriented paradigm itself.

This description is followed by views on how to apply the paradigm from analysis to design to implementation. A listing of the benefits and drawbacks of the object-oriented approach follows how to apply the paradigm. The chapter concludes with examples of successful applications of the object-oriented software development strategy.

2.2 The Software Crisis

"The software crisis encompasses problems associated with how we develop software, how we maintain a growing volume of existing software, and how we can expect to keep pace with a growing demand for more software" (42:23). In this vein, Brooks likens a software project to a werewolf: "it is usually innocent and straightforward, but is capable of becoming a monster of missed schedules, blown budgets, and flawed products" (7:10).

A cause of the software crisis is the complexity of software (3, 7). Booch describes complex programs as "industrial-strength software". "Stated in blunt terms, the complexity of such systems exceeds the human intellectual capacity" (3:3). Booch argues that there are five attributes of complex systems:

1. The system will possess a hierarchy. The system will be composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of primitive components is reached.

2. The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.
3. The linkages between components will change less than the components themselves.
4. Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements.
5. A complex system that works is invariably found to have evolved from a simple system that worked.

(3:10-11)

Another "essence" of modern software systems according to Brooks is "changeability" (7). "The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product" (7:12).

The consequence of this "changeability" is software maintenance. The cost of software maintenance is significant. According to Somerville, "Evidence from existing systems suggests that maintenance costs are, by far, the greatest cost incurred in developing and using a system" (54:536).

The previous discussions highlight the fact that developing software and changing it after it is built is a formidable problem. Is there a "silver bullet" to slay the software project "werewolf"? While it may not be the "silver bullet" that provides an order of magnitude improvement in reliability, productivity or simplicity, Brooks throws his support behind the object-oriented paradigm (7:10,14).

2.3 The Object-Oriented Paradigm

The object-oriented paradigm is not a mature technology (11:156). Many authors do not agree on what the elements of the paradigm are. Fortunately, there were a number of elements of the object-oriented model that did appear most often

in the various works that I read: the object, the class, the relationship between classes and interobject communication.

2.3.1 Objects What is the object-oriented way of making software? "Simply stated, object-oriented development is an approach to software design in which the decomposition of a system is based upon the concept of an object" (4:5). "Objects are abstractions of like instances of any concept in the real world" (51:67). "Whereas a procedure models an action, an object models some *entity* in the problem domain, encapsulating both data about that entity and operations on that data" (50:95). "A design is complete when every object that is referenced has been defined and every operation is defined" (26:27).

2.3.2 Classes A class is a description of the data and the operations on the data that make up the objects of the class. "An object is said to be an *instance* of its class" (47:2). "Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the "essence" of the object, as it were" (3:93).

More generally, to borrow Coad and Yourdon's quote from Webster's dictionary, a class is "A number of people or things grouped together because of certain likenesses or traits" (10:52). The main point is that a "class" is a description of the set of objects that share the same characteristics.

In addition to the description of the objects, the class may also have data associated with it. The class data can be accessed by all instances (objects) of the class (3, 37, 60).

2.3.3 Relationships Between Classes There are two types of relationships between classes: inheritance and composition.

2.3.3.1 Inheritance Rumbaugh defines inheritance as "the sharing of attributes (data) and operations (on the data) among classes ... A class can be defined broadly and then refined into successively finer *subclasses*. Each subclass incorporates, or *inherits*, all of the properties of its *superclass* and adds its own unique properties" (47:3).

The previous definition of inheritance brings up a pertinent point. It is important to keep in mind that a class is nothing more than a description. The words "object" and "class" are sometimes used interchangeably, or at least confusingly, as evidenced by Rumbaugh's definition. What "sharing ... among classes", actually means is that *objects* of that subclass will possess the same data and allow the same operations on that data that an *object* of the superclass will. Also, objects of the subclass will have additional data and/or operations that the subclass adds on top of what the superclass specifies.

Inheritance is used to portray generalization and specialization (10:15). Inheritance can best be thought of as an "is a" or "is a kind of" relationship between classes (10:79). For example, a "truck" (subclass) can be thought of as a kind of "vehicle" (superclass). The subclass is a more specific class than the more general superclass that it is derived from. For example, a truck object will have the same attributes and operations that a vehicle object has in addition to attributes and operations common only to trucks.

The discussion thus far has described a relationship where one class inherits properties from only one other class. "Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. This permits mixing of information from two or more sources" (47:65).

2.3.3.2 Composition The second type of relationship between classes is the "composition" relationship. This means that one or more of the parts of a class will be an object of another distinct class. Coad and Yourdon term this a "Whole-

Part" relationship (10:91). A simple example would be that an aircraft consists of (or "has parts" or simply "uses") an engine, where "aircraft" and "engine" are distinct classes of objects (10:92).

2.3.4 Communication Between Objects An object-oriented system functions through interobject communication or "message passing". A major part of constructing an object-oriented system is deciding what operations will be provided by each object and which other objects will need to use the operations (4, 10, 34).

2.3.5 Terminology The object-oriented paradigm has a terminology all its own. Unfortunately, there is no standard vocabulary. "Object-oriented design methodologies are still in their early stages. Like the various object-oriented programming languages, terminology for the object-oriented mechanisms differs among methodologies" (60:189). The purpose of this section is to present the various terms of the object-oriented paradigm and their aliases. Terminology associated specifically with the C++ programming language is included.

The first term listed will be the one that is used most frequently in the remainder of this thesis. The word will be followed by any aliases of the term. These aliases may appear within verbatim quotations from authors included within the rest of the thesis.

abstract class: "A class that has no instances. An abstract class is written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its (typically) incomplete methods" (3:512).

attribute, data member, field, instance variable, member object, slot: "A property or characteristic of an object" (60:262). "An attribute is some data (state information) for which each object in a class has its own value" (10:119).

base class, ancestor class, parent class: With respect to an inheritance relationship, the superclass class is the more general of the two (or more) classes. The "parent" of the relationship.

class, type: "A set of objects that share a common structure and a common behavior" (3:513). A class is a description or a template of the objects that are members of the class.

composition, aggregation, Whole-Part, using: A type of relationship between classes in which one class includes an object of another class. An object of the class that uses the other class will have as an attribute an object of the class being used.

derived class, descendant class, subclass, child class: With respect to an inheritance relationship, the derived class is the more specialized of the two classes.

framework: A collection of classes that all relate to a specific problem domain. They are specifically designed to be reused for applications within the problem domain for which they were constructed.

inheritance, generalization, specialization, is a, is a kind of: A type of relationship between classes where one class is derived from one or several more general classes. The derived class has the same characteristics of the more general class(es) and may add its own attributes and/or methods to the resulting class description.

interface, contract, protocol, signature: The set of methods that are provided by an object. The protocol constitutes the outside view of the object with respect to the whole system.

message: The act of one object using a method of another object. This usually takes the form of a procedure or function call.

method, member function, operation, service: "A specific behavior that an object is responsible for exhibiting" (10:143). Methods may modify the state of the object.

object, entity, instance: "Objects are (run time) entities that encapsulate within themselves both the data describing the object and the instructions for operating on that data" (60:269).

2.4 Major Principles of the Object Oriented Paradigm

As was the case with deciding what the elements of the object-oriented paradigm are, different authors present varying opinions as to what major principles are embodied within the object-oriented paradigm. The two principles most often mentioned were abstraction and encapsulation.

2.4.1 Abstraction "Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant. Abstraction must always be for some purpose, because the purpose determines what is and what is not important" (47:16).

The concept of an object provides abstraction in the object-oriented paradigm. An object is a product of the principle of data abstraction: "The principle of defining a data type in terms of the operations that apply to the objects of the type, with the constraint that the values of such objects can be modified and observed only by the use of the operations" (10:14). This abstraction (the object) serves as a basis for organization of thinking and of specification of a system's responsibilities (10:14).

"An abstraction focuses on the outside view of an object, and so serves to separate an object's essential behavior from its implementation" (3:40). When trying to gain an understanding of a system, the behavior of the object (the "what") is what is important while the details of how the methods are implemented (the

"how") are not. The behavior is reflected in the set of operations provided by the object that are used to modify the data contained in the object.

2.4.2 Encapsulation "*Encapsulation* (also *information hiding*) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects" (47:7). "Information hiding and abstraction are two sides of the same coin" (39:90). "Abstraction and encapsulation are complementary concepts: abstraction focuses on the outside view of an object and encapsulation prevents clients from seeing its inside view, where the behavior of the class is implemented" (3:45).

Information hiding is built into the object-oriented paradigm through the class construct. "The class construct supports information hiding through the separation of the class interface and the class implementation" (28:51). The concept of the class formalizes the idea that "no parts of an object-oriented program can operate directly on an object's data. Communication among a set of objects occurs exclusively through explicit messages" (60:36).

Mullin provides a good example of information hiding: "The watch is an object, one that satisfies my requests for current time. It does not need me to tell it how to do its job!" (37:21).

2.5 How To Apply the Object-Oriented Paradigm

"At the most general level, three phases to the (software) lifecycle are agreed upon: 1) analysis, 2) design and 3) construction/implementation" (25:144). Like other software development methodologies, the object-oriented software development process also has analysis, design and implementation phases. However, in contrast to more traditional methods, the object-oriented development process is iterative and the same constructs are used and expanded upon in each successive phase of development (28:41).

The object-oriented software development cycle is a "unifying paradigm" (28). The results from each phase are used directly in the following phases. This is in contrast to the classical software development lifecycle in which the results of the analysis (eg. data flow diagrams) have to be translated into some other form in the design phase (eg. structure charts).

Another way that the object-oriented software development lifecycle differs from more traditional approaches is that the developer is expected to iterate freely through the stages in the lifecycle. Booch terms this "round-trip gestalt design" (3). "This style of design emphasizes the incremental and iterative development of a system" (3:188).

The line between the phases is very thin at best. "The reason for this blurring is that the items of interest in each phase are the same: objects. A second reason is that the object-oriented development process is iterative" (28:41). Another possible reason is that while object-oriented programming is well developed, techniques for object-oriented analysis and design are not (10:156).

The next three sections are devoted to "drawing the line" between the three phases of object-oriented software development. The first covers analysis, the next section discusses design and the last explores object-oriented programming and the C++ programming language. The fourth part of this section briefly covers the various notations used for object-oriented software development.

2.5.1 Object-Oriented Analysis (OOA) "Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain" (3:37). "The purpose of object-oriented analysis is to model the real world system so that it can be understood" (47:148). "Then the analyst focuses in on those matters pertinent to his work, namely, describing the responsibilities of the system under consideration" (10:9-10). This analysis focuses on the "what" of the system and not the "how".

"The analysis model serves several purposes: It clarifies the requirements, it provides a basis for agreement between the software requestor and the software developer, and it becomes the framework for later design and implementation" (47:148). A bibliography of the sources that I used in researching OOA are detailed in appendix A. The following activities are a synthesis of their methods: 1) Identify the classes/objects of the problem space, 2) Identify the relationship between classes, 3) Identify the attributes and methods of each class/object, and 4) Specify the inter-object communication. All of the activities may overlap. Table 2.1 shows how the strategy of each author relates to the synthesized list.

2.5.1.1 Identify Classes/Objects The purpose of this stage is to come up with the classes/objects that make up the problem domain. The classes/objects are identified with respect to the specific problem being modeled. "With OOA, an analyst *studies* the overall problem domain, *filters* that problem domain understanding to just those aspects which are within the system's responsibilities, and *models* it accordingly" (10:53). The end result should be a model of the real world system (47:148).

Various authors provide a number of tips on how to find classes/objects. Coad and Yourdon (10) offer a checklist of where to look and what to look for:

Identify Classes & Objects	Bailin — Steps 1,2,4,5, & 7 Booch — Step 1 Coad & Yourdon — Object Layer Henderson-Sellers — Steps 1 & 2 Rumbaugh — Object Model (step 1)
Identify Attributes and Methods	Bailin — Steps 1,2,4,5, & 7 Booch — Steps 1 & 2 Coad & Yourdon — Attribute Layer Coad & Yourdon — Service Layer Coad & Yourdon — Class & Object Templates Henderson-Sellers — Step 2 Rumbaugh — Object Model (step 4) Rumbaugh — Dynamic Model Rumbaugh — Functional Model Shlaer — Steps 1,2, & 3
Identify the Relationships Between Classes	Bailin — Steps 1,2, & 4 Booch — Step 4 Coad & Yourdon — Structure Layer Henderson-Sellers — Steps 6 & 7 Rumbaugh - Object Model (steps 3,5 & 6) Shlaer - Step 1
Specify Interobject Communication	Bailin — Step 3 Booch -Step 3 Coad & Yourdon — Attribute Layer Coad & Yourdon — Service Layer Coad & Yourdon — Class & Object Templates Henderson-Sellers — Step 3 Rumbaugh — Object Model (steps 3,4, & 6) Rumbaugh — Dynamic Model Rumbaugh — Functional Model Shlaer — Steps 2 & 3

Table 2.1. Comparison of OOA Techniques

Where to Look:

1. Observe the system first-hand.
2. Actively listen to problem domain experts.
3. Check previous OOA results from similar problem domains.
4. Check other similar systems.
5. Read, read, read (the problem statement and the customer request).

What to Look For:

1. Structures within the problem domain.
2. Other complete systems that the system interacts with.
3. Devices that the system interacts with.
4. Things or events remembered.
5. Roles played.
6. Operational procedures.
7. Sites or locations the system must be aware of.
8. Organizational units.

Shlaer and Mellor (51) offer the following list of what to look for:

1. Tangible things.
2. Roles of things.
3. Specifications or quality criteria.
4. Useful aggregations of equipment.
5. Steps in a manufacturing process.

Rumbaugh (47:153) and Booch (3, 4) recommend trying to identify classes/objects by picking out the nouns in the problem statement. The nouns would represent candidates for classes/objects of the system.

2.5.1.2 Identify the Relationship Between Classes The purpose of this phase is to specify the structure inherent in the system. In this activity, the inheritance and composition relationships between classes is identified. The process of recognizing and differentiating between an inheritance and a composition relationship is central to this phase.

The first thing to do is look for common structures in the problem space in order to find inheritance relationships. "Inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by refining existing classes in specialized subclasses (top down)" (47:163). Coad and Yourdon recommend examining every class as a potential superclass or subclass with respect to the problem domain (10:84,86).

The next step is to try to look for composition relationships. Coad and Yourdon advocate looking for the following variations in the problem domain: 1) Assembly-Parts, 2) Container-Contents, and 3) Collection-Members (and its different varieties) (10:93).

It is not always clear as to when to specify an inheritance versus a composition relationship. Booch offers a general rule of thumb: "if an abstraction is greater than the sum of its component parts, then using relationships are more appropriate. If an abstraction is a kind of some other abstraction, or if it is exactly equal to the sum of its components, then inheritance is a better approach" (3:116).

An example of composition might make this rule of thumb easier to understand. A telephone can be modeled by a relationship where the telephone class inherits phone pads, microphones, and speakers (multiple inheritance) or the telephone class can simply use the component classes (3:116). In this instance the sum of the parts is greater than the whole — putting the pieces together in a telephone is greater than simply each of the pieces lying in a group on a table. Also, it doesn't make any sense to say that a telephone "is a kind of" phone pad (or any other component) (3:116).

2.5.1.3 Specify Class/Object Attributes and Methods The "identity" of each class/object is defined in this stage. The state and behavior embodied by each individual object must be fully specified. This specification includes the attributes of each object and the methods used to modify the attributes.

"Attributes describe values (state) kept within an Object, to be exclusively manipulated by the Services of that Object" (10:120). "Attributes are properties of individual objects, such as name, weight, velocity or color" (47). In order to identify possible attributes, Coad and Yourdon suggest asking the following questions from the perspective of a single object:

1. "How am I described in general?"
2. "How am I described in this problem domain?"
3. "How am I described in context of this system's responsibilities?"
4. "What do I need to know?"
5. "What state information do I need to remember over time?"
6. "What states can I be in?"

Attributes can also be objects of another class. This is a composition relationship between the class of the object and the class of the object being used as an attribute. Thus, this phase can lead back to phase two. "If the independent existence of an entity is important, rather than just its value, then it is an object" (47).

The specification of the methods of the class/object come next. "One theme underlying Object-Oriented Analysis is that eventually the analyst must provide a detailed description of a system's processing and sequencing requirements" (10). This begins with identifying "primitive" methods provided by each object.

Booch lists three common kinds of operations: a modifier, a selector and an iterator (3). A modifier alters the state of the object. A selector accesses the state

but does not modify it. The iterator permits all parts of an object to be accessed in some well-defined order.

There will be methods that need the methods of other objects or require that some action(s) be performed prior to using a method. While this obviously leads to phase 4 — interobject communication — it is important to specify these requirements in terms of the method that needs them. This fact is recognized by all of the authors but again, they give different ways of detailing the requirements.

Most of the authors advocate some type of state table or "state model" to depict the lifecycle of an object (1, 10, 47, 51). This state model is also used in some cases to further identify methods that will be needed or provided (47, 51).

Once the methods are identified, they must be explained in some manner so that they can be implemented. Coad and Yourdon advocate "Service Charts" and "Object State Diagrams" which are intended to portray services and state dependent behavior (10:157). Shlaer recommends data flow diagrams (51:66). Rumbaugh recommends using traditional data flow diagrams, natural language, mathematical equations and/or pseudo code (47:179).

2.5.1.4 Interobject Communication Interobject communication goes by many names: message connection (10), association (47), relationships (51) and data flow between entities (1). The objective is to specify what methods are needed by each object that are provided by other objects in the system.

As was mentioned in the previous section, this phase overlaps with phase 3. Specifying what methods are required by an object from other objects often leads to the identification of new methods. Thus, a state model (of some kind) can also be used in this phase as well. "The processes required to drive an object or relationship through its lifecycle are derived from the actions of the state model" (51:66).

2.5.2 Object-Oriented Design (OOD) "During analysis, the focus is on what needs to be done, independent of how is done. During design, decisions are made about how the problem will be solved, first at a high level, then at increasingly detailed levels" (47:198). The focus now is on the "how" given that the analysis stage has defined the "what". This section is devoted to OOD and will consist of three parts: 1) a presentation of two methods of OOD, 2) a discussion of polymorphism, and 3) a presentation of some guidelines to use in making a good design.

Bertrand Meyer remarked in 1988 that "The literature on object-oriented design (as opposed to just programming) is sparse" (34:334). The situation is very much the same today. The majority of the relatively few sources that I was able to find didn't attempt to differentiate between analysis and design, included analysis activities in their description of OOD and/or simply added "and then implement the objects" as a final step in their particular process (3, 4, 22, 50). A possible explanation for this lack of material on object-oriented design comes from Henderson-Sellers: "The design stage is perhaps the most loosely defined since it is a phase of progressive decomposition toward more and more detail and is essentially a creative, not a mechanistic, process" (25:144).

Two books that did present a definite design process were "Object-Oriented Design" by Peter Coad and Edward Yourdon and "Object-Oriented Modeling and Design" by James Rumbaugh and others. The views of these authors will be presented in the first part of this section.

One important term of the object-oriented paradigm has not yet been mentioned: polymorphism. "In general, polymorphism means the ability to take more than one form. In an object-oriented language, a polymorphic reference is one that can, over time, refer to instances of more than one class" (28:45). The topic of polymorphism is discussed in the second part of this section.

There were more sources of advice on what makes a good design than sources of how to construct a design. The third part of this section details proposed criteria

for measuring a good design gathered from various authors.

2.5.2.1 Two Methods of Object-Oriented Design This section presents two methodologies for performing object-oriented design. The first method is advocated by Peter Coad and Edward Yourdon (11). The second method is advanced by James Rumbaugh et. al. (47). Both methods assume that OOA has been done prior to design and that the products of the analysis form the basis for the design.

1.5.2.1.1 The Coad and Yourdon Method of OOD The Coad and Yourdon method is based upon the construction of four components: 1) the Problem Domain Component, 2) the Human Interaction Component, 3) the Task Management Component, and 4) the Data Management Component (11:25).

"In OOD, the OOA results fit right into the Problem Domain Component (PDC)" (11:36). The idea is to use the results of OOA and add to them within the constraints of Coad and Yourdon's method. However, "These additions do not mean it is time to hack up analysis results, whip up a little magic, and then suddenly "poof" away into design" (11:36).

Coad and Yourdon offer a number of criteria to use when adding to the OOA results during the construction of the PDC:

1. Reuse design and programming Classes — look for opportunities to reuse existing "off-the-shelf" classes.
2. Group problem domain specific Classes together — you can add a class simply to group classes together within a Class library (in lieu of a more sophisticated way to do this).
3. Establish a protocol by adding a generalization of a class — add a class to formalize the interface of derived classes.

4. Accommodate the supported level of inheritance — it may be necessary to revise the various inheritance relationships if multiple inheritance or any form of inheritance is not supported.
5. Improve performance — may need to rearrange or combine classes to reduce message traffic.
6. Support the Data Management Component — to support the Data Management Component, each object must know how to store itself or must send itself to another object designed to save objects
7. Add lower-level components — mostly a matter of convenience or to aid in understandability.
8. Don't modify just to reflect team assignments — don't split up related classes between different software development teams.
9. Review and challenge the additions to OOA results — whenever and wherever possible, preserve the problem-domain-based organization established by OOA results.

(11:39-48)

The Human Interface component captures how a human commands the system and how the system presents information to the user (11:56). One of the driving forces behind the object-oriented paradigm has been the construction of user interfaces (60:9). Coad and Yourdon devoted a stage in their process exclusively to examine the user interface in detail.

The strategy to design the Human Interface Component consists of the following:

1. Classify the humans — who uses the software?
2. Describe the humans and their task scenarios — what does the user want to do with the system?

3. Design the command hierarchy — what commands will be offered and how will they be presented?

4. Design the detailed interaction — design the interface with good "human interaction" principles in mind.

5. Continue to prototype — the best way for a user to make an evaluation is to let them use a representation of the real thing.

6. Design the HIC classes — add the classes used specifically for implementing the HIC (if they are not there already).

7. Design, accounting for Graphical User Interfaces (when applicable) — design around one if it is available to use: Macintosh, Windows, Presentation Manager, X Windows, and Motif.

(11:57-67)

The motivation for the Task Management Component is the determination of concurrency within the system. Elements such as external devices, external inputs, human interfaces, and the multiprocessing capabilities of the machine are considered in this phase. The word "task" indicates concurrent behaviors in the system. The strategy for determining tasks are as follows:

1. Identify event-driven tasks — a task may be designed to trigger upon the receipt of a certain event.

2. Identify clock-driven tasks — these tasks are triggered at a specified time interval.

3. Identify priority tasks and critical tasks — high priority tasks are those that may need to be separated out in order to get the Service done within an urgent time constraint. A critical task affects the continued operation of the system itself.

4. Identify a coordinator — when multiple tasks exist within the system, it may be necessary to add another task that coordinates them.
5. Challenge each task — keep the number of tasks to a minimum.
6. Define each task — define each task by what it is, how it coordinates, and how it communicates.

(11:73-76)

"The Data Management Component (DMC) provides the infrastructure for the storage and retrieval of objects from a data management system. The Data Management Component isolates the impact of data management scheme, whether flat file, relational, object-oriented (or some other one)" (11:80).

The first step in making the DMC is to design the data layout of objects with reference to the data management scheme that will be used. The second step is to define the services needed to actually store and retrieve the objects given the data layout.

1.5.2.1.2 The Rumbaugh (et al) Method of OOD There are two major steps to Rumbaugh's method: system design and object design. "System design is the first design stage in which the basic approach to solving the problem is selected. The system architecture is the overall organization of the system into components called subsystems. By making high-level decisions that apply to the entire system, the system designer partitions the problem into subsystems so that further work can be done by several designers working independently on different subsystems" (47:198-199).

There are eight decisions made in the system design phase:

1. Organize the system into subsystems — group together aspects of the system that share some common property.

2. Identify concurrency inherent in the problem — identify which objects must be active concurrently and which objects have activity that is mutually exclusive.
3. Allocate subsystems to processors and tasks — choose a software or hardware implementation of the subsystem and allocate subsystems to processors. Must keep performance and low interprocess communication in mind.
4. Choose an approach for management of data stores — choose between files and/or databases.
5. Handle access to global resources — identify global resources and determine mechanisms for controlling them. Global resources include things like tape drives, processors, disk space and access to shared data.
6. Choose the implementation of control of software — choose between procedure-driven sequential, event-driven sequential and concurrent control. Keep in mind the implementation language and operating system.
7. Handle boundary conditions — decide how to start/initialize the system, how to terminate and what to do in case of an unplanned termination.
8. Set trade-off priorities — make a decision as to what gets priority during the development. Decide between such factors as speed, memory available, portability, functionality, cost and time available.

(47:199-211)

"The analysis phase determines what the implementation must do, and the system design phase determines the plan of attack. The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations" (47:227).

During object design, the designer must perform the following steps:

1. Combine the three models to obtain operations on classes — this step intends to build upon the analysis results of their particular model (which consists of three parts). While the majority of operations should have been identified in analysis, it is perfectly fine to add more in the design phase.
2. Design algorithms to implement operations — choose algorithms that minimize the cost of implementing them. You may define new classes and operations as necessary.
3. Optimize access paths to data — may restructure class organizations to optimize access, add attributes that store frequently calculated values or you may rearrange execution order for efficiency.
4. Implement control for external interactions — implement the control strategy decided upon in the system design phase.
5. Adjust class structure to increase inheritance — look for more opportunities to derive commonality between classes.
6. Design associations — Rumbaugh's approach is deeply rooted in database theory. The primary tool of the analysis is an entity-relationship diagram (of sorts) showing associations between the entities (objects) in the system. These associations most often become message passing between the objects in question.
7. Determine object representation — decide whether to use primitive types (integer, string, real, etc.) or implementation as an object. SSAN is a good example. Do you make it an object or simply implement it as a string of 9 characters?
8. Package classes and associations into modules — group related pieces of software together in one physical location (file).

(47:228-249)

2.5.2.2 Polymorphism In general, polymorphism means the ability to take on many forms. Polymorphism, with respect to object-oriented design, concerns inheritance and late binding (3:104). "This refers to the ability of an entity to refer at run-time to instances of various classes" (34:224).

There is a difference between overloading and polymorphism. Polymorphism is a run-time, dynamic, phenomenon. With overloading, the compiler can statically determine which method to call. The compiler determines the method to call based upon the parameter profile of the method. The profile consists of the number and types of the method's arguments and the types the method may return. If the compiler can find a match, then it is considered overloading and not polymorphism (3).

Inheritance is the mechanism that drives polymorphism. "The "is a" nature of inheritance is tightly coupled with the idea of polymorphism. The idea is that if Y inherits from X, Y is an X, and therefore anywhere that an instance of X is expected, an instance of Y is allowed" (28:45). If the program does not explicitly reference a method of X, the compiler cannot determine whether a reference to a method of X actually applies to X or to a method of Y. This reference must be resolved at run-time.

2.5.2.3 Measures of a Good Design Various authors present guidelines or rules to use in order to help ensure a good design (3, 11, 16, 26, 28, 32, 34, 47, 61). All of the authors either explicitly mentioned or based their rules upon the measures of coupling and cohesion.

"Coupling is the "interconnectedness" between pieces of an OOD" (11:129). Coupling refers not only to the number of interconnections but also to the complexity of the interconnections (11, 34). The goal is to have the least amount of coupling between abstractions (34:18-20). There should be as little message passing between classes as possible. In addition, if two classes do communicate, they should pass as

little information as possible (34:20).

"A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of functional relatedness" (54:189). "Cohesion measures the degree of connectivity among the elements of a single module (and for object-oriented design, a single class or object)" (3:124). Cohesion can be used to evaluate the methods and the overall structure of a class or object (16:145). The goal is to have abstractions that are highly cohesive.

David Embley provides the most comprehensive treatment of the application of the principle of cohesion to a design (16). His article provides rules to use to evaluate the quality of abstract data types (ADTs) written in Ada. While an ADT is slightly different from a class (56:13), the same basic principles can be applied to both. Embley's methods are based upon comparing the types of the ADT with the operations on those types. This compares to examining the attributes of a class with respect to the methods provided by the class.

The following recommendations have been extracted from Embley's article and modified so that they apply to the classes, attributes and methods of an OOD:

1. Draw a graph whose nodes correspond to the attributes and methods of the class. Draw the links of the graph from each method to each attribute that the method uses. If the resulting graph is disjoint, then you should consider splitting the class into two (or more) separate classes. This is because there are distinct sets consisting of methods and attributes used by those methods that might have no relation to each other.
2. Try not to include a more general class within the definition of another. These are candidates for an inheritance relationship.
3. Do not nest classes within others. Nested class definitions are included in another class and can only be used by the class that contains it.

4. Try to create classes where the methods either use a single attribute or use all the attributes. Classes that do not exhibit this behavior may be candidates for further decomposition.

The following list was compiled from several sources. Most of these relate in some way to the principles of coupling and cohesion. These principles are included to give a more detailed view of how to arrive at a good design:

The Responsibility-Driven Approach: when designing a class, focus on what the class is responsible for remembering and providing, not on the details of exactly how the information is stored. This should ensure that the internal structure of the object is not visible to users of the object (61).

The "Law of Demeter": each method can send messages to only a limited set of objects: objects included as arguments to the message, the object of which the method is a part, or to objects that are components of the object that the method is a part of (32, 33).

Sufficiency: "the class captures enough characteristics of the abstraction to permit meaningful and efficient interaction" (3:124). Does the object provide enough methods to effectively use it?

Completeness: "the interface of a class captures all of the meaningful characteristics of the abstraction" (3:125). Does the object contain a complete set of methods?

Primitiveness: the methods of classes must be primitive. "Primitive operations are those that can be efficiently implemented only if given access to the underlying representation of the abstraction" (3:125). The concept of primitiveness is included primarily to prevent the concept of completeness from being carried too far. If a method (that may have been added for completeness' sake) can be done by using simpler methods already available, then the method should not be offered.

Inheritance guidelines: each subclass should be developed as a specialization of the superclass. All public methods of the superclass should become part of the public part of the subclass. The root class of an inheritance structure should be an abstract model of the target concept (28:54). There are differing opinions concerning subclass visibility of attributes inherited from the superclass. They range from no direct visibility (53) to total visibility (33).

Method guidelines: each method of a class should use at least one of the attributes of the class. A method should be public only if it is meant to be available to users of the class. Methods of one object should not directly access the attributes of a different class (28:54).

Clarity of design: Use a consistent vocabulary. The names in the model should correspond to the names a reader would expect for that component. Use consistent names for similar methods (11:141-143).

Generalization-Specialization depth: Do not create levels of specialization simply for the sake of doing it. Have a reason for creating each level in the inheritance hierarchy (11:143).

Simplicity: this applies to classes and methods. Excessive attributes are an indication of poor factoring. Methods should not have too many parameters. "If a message requires more than three parameters, on average, something is wrong" (11:145). The implementation of the methods should be small as well. "In general, if the method looks like a block-structured program, the classes have been poorly chosen" (11:145).

Critical Success Factors: evaluate the design on the basis of its potential for reuse, readability and performance (11:147).

Class hierarchies should be deep and narrow: "A class hierarchy having one superclass and 27 subclasses is much too shallow. A shallow class hierarchy is evidence that change is needed" (26:29).

Factor implementation differences into subclasses: by including methods and attributes that are entirely implementation dependent into subclasses, the superclasses become easier to reuse (26:35).

"Real world constraints always bastardize the most elegant design. Inevitably, your design will be compromised to accommodate language shortcomings or performance demands or a trade-off between reusability and development costs ... The idea is to take a hard look at such compromises in light of "good design" principles and whether the perceived constraint actually necessitates the change (11:148).

2.5.3 Object-Oriented Programming (OOP) "Simply stated, object-oriented programming deals with the manipulation of objects" (21:2). "Object-oriented programming is a methodology for creating programs using collections of self-sufficient objects that have encapsulated data and behavior and which act upon, request, and interact with each other by passing messages back and forth" (60:270).

"In the software lifecycle, object-oriented programming concentrates on the design and implementation stages of software engineering ... because object-oriented programming encompasses both design and implementation, it tends to blur the distinction between the two. As Meyer (35:63) points out, this is an advantage since design and implementation are essentially the same activity: constructing software to satisfy a certain specification. The only difference is the level of abstraction: During the design certain details are left unspecified, but in an implementation everything is specified" (23:70-71).

"Object-orientation changes the focus of the programming process from procedures to objects" (60:iv). Object-oriented programming is an alternative way to write software instead of using a procedurally based approach. It will not ensure that the resulting software will be any better than a procedurally based program.

"A truly object-oriented design can be directly implemented only in an object-oricnted language" (28:56). "An "object-oriented programming language" means

that the language has mechanisms that support the object-oriented style of programming well. A language *supports* a programming style if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style" (56:10).

Opinions differ slightly, but in order to support the object-oriented style of programming, the language must allow: 1) "objects" in the sense that they are data abstractions with an interface of named operations and a hidden local state, 2) the concept that objects belong to a "class", 3) inheritance, and 4) polymorphism (3, 21, 47, 56, 59, 60).

2.5.4 Notation Various notations have been introduced for use in the object-oriented software development lifecycle (3, 10, 25, 34, 37, 47, 58). Each notation was devised to capture the different elements and relationships in an object-oriented design (according to the author's point of view).

The notations advocated by Wasserman, Coad and Yourdon, and Booch are presented in figures 2.1 through 2.5.

2.6 Benefits and Drawbacks of the Object-Oriented Approach

The are many benefits as well as a few drawbacks to the object-oriented paradigm.

2.6.1 Benefits of the Object-Oriented Approach The object-oriented paradigm offers the following benefits: 1) it offers a way to manage complex software development efforts, 2) it provides a "seamless" way to perform analysis, design and implementation, 3) it promotes reusability, and 4) it promotes maintainability and extensibility.

2.6.1.1 Manages Complexity The object-oriented paradigm attacks complexity mainly through: 1) the two principles of abstraction and encapsulation embodied in the paradigm, and 2) through the idea of inheritance to depict the

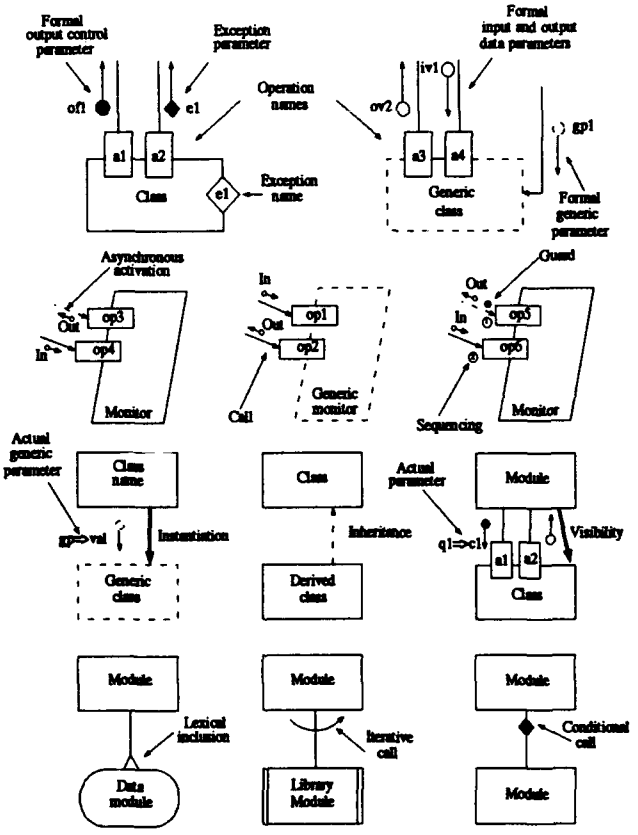


Figure 2.1. The Object-Oriented Structured Design Notation (58)

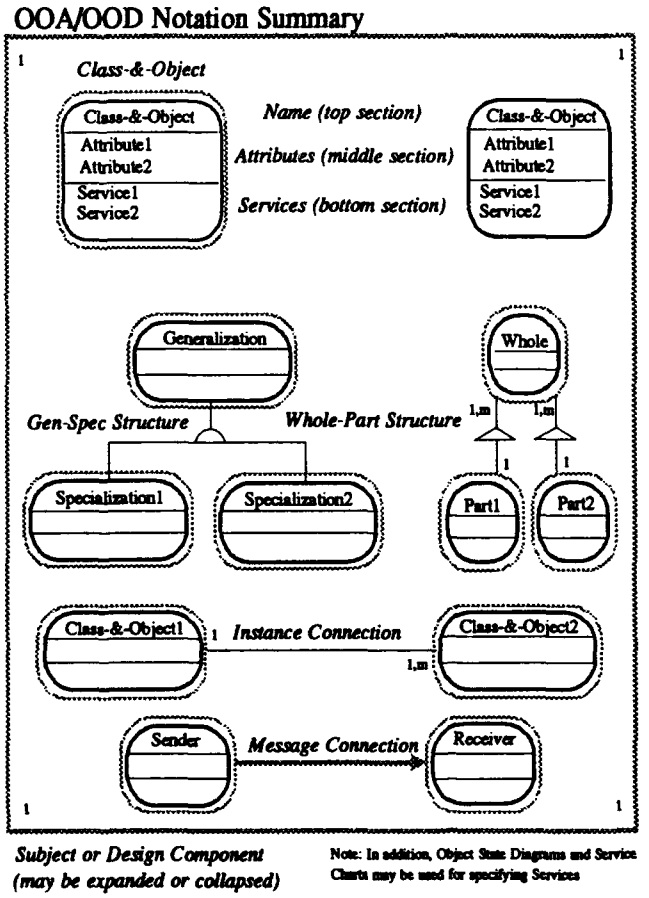


Figure 2.2. Coad and Yourdon's OOA/OOD Notation (10, 11)

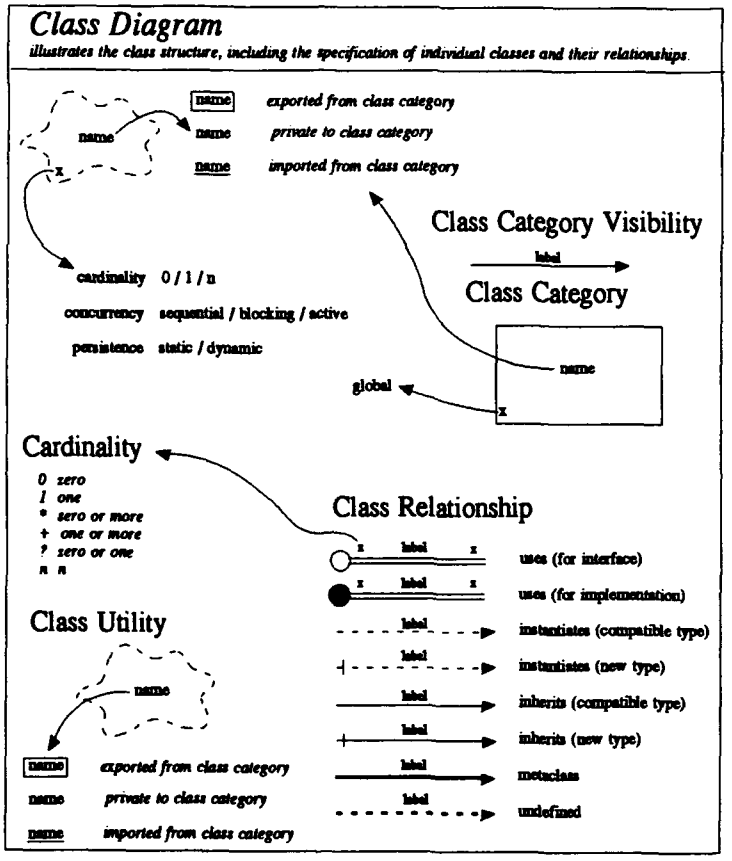


Figure 2.3. Booch Class Diagram (3)

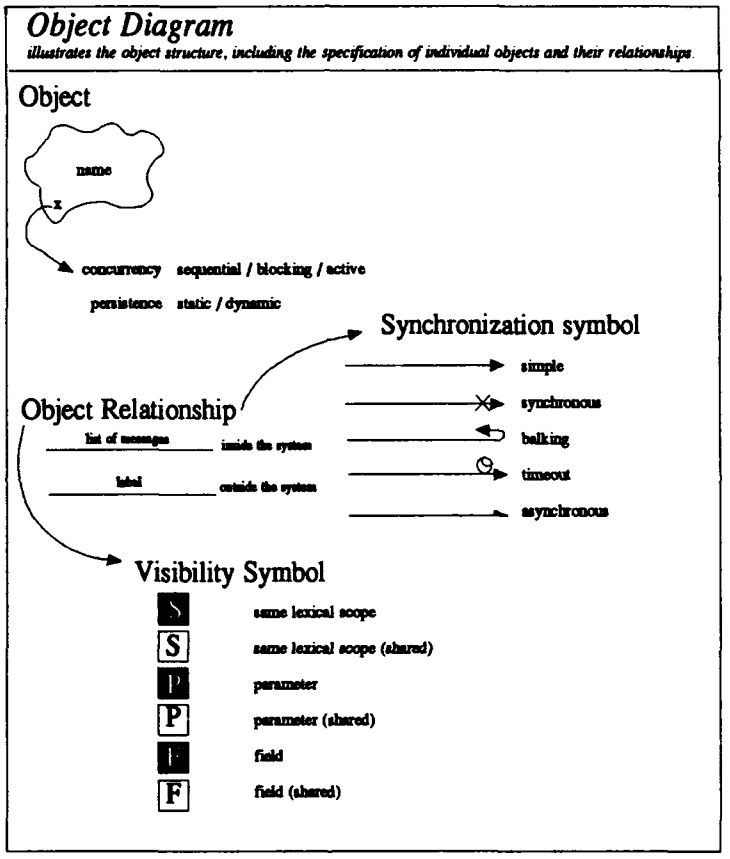


Figure 2.4. Booch Object Diagram (3)

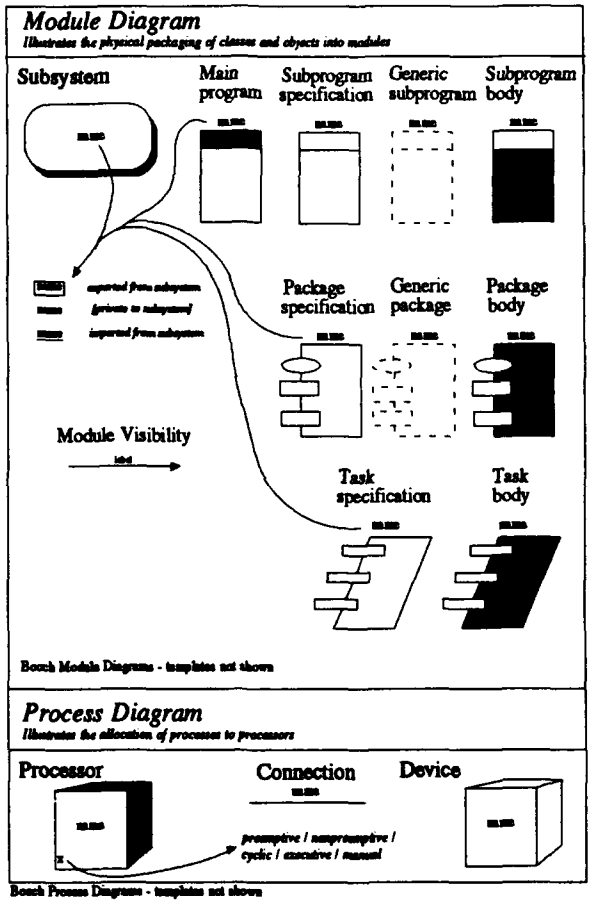


Figure 2.5. Booch Module and Process Diagrams (3)

hierarchy in a system (3:39,45). "The most important point scored by OO techniques is, however, that they are natural and intuitive" (29:15).

The concept of an object is an abstraction of a real world entity in the problem domain that communicates with other objects only through specifically provided methods. Encapsulation is exemplified in an object by the fact that the implementation of the object's state and methods is hidden.

The object-oriented approach to building software can be applied to all five of Booch's five attributes of complex systems (given in section 2.2, from (3:10-11)):

1. The system will possess a hierarchy. The system will be composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of primitive components is reached.

— The inheritance and composition relationships between classes directly model hierarchy in a system. "By identifying these hierarchies in our design, we greatly simplify our understanding of the problem" (3:54).

2. The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system.

— Objects are the components of an object-oriented system. The objects will correspond to their counterparts in the "real world". Their "primitiveness" is driven by the problem being solved.

3. The linkages between components will change less than the components themselves.

— This is addressed through abstraction and encapsulation represented by the object. An object is altered only by the methods provided by the object (the linkage between components) while the implementation of the object is hidden from the user(s) of the object. Therefore, any changes to the implementation are localized to the object. There will be no "popcorn" effect if the innards of an object change (11:129).

4. Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements.

— Again, the idea of inheritance addresses this aspect of complexity. Inheritance allows a designer to express the commonality found in a system (3:56).

5. A complex system that works is invariably found to have evolved from a simple system that worked.

— Object-oriented software construction is primarily a

"bottom-up" approach (34:325). Simple objects can be implemented and tested before being "plugged in" much like a "software IC" as a part of a more complex system (12).

2.6.1.2 Seamless Paradigm The object-oriented software development process can be used from analysis to design and through to implementation. "The design philosophy of the object-oriented paradigm takes a modeling point of view. This allows the designer to work with one approach which begins in the problem domain and transitions naturally into the solution domain" (28:60).

"It is easier to design and implement object-oriented applications because the objects in the application domain correspond directly to objects in the software domain. This one-to-one correspondence eliminates the need to translate a design to a less natural programming language representation, even though most programmers have been trained to do this translation" (60:45).

2.6.1.3 Promotes Reusability The object-oriented paradigm provides support for reusability through classes and the relationships between them: inheritance and composition. "Every time an instance of a class is created, reuse occurs. This means more than a declaration of a variable of a specific type. The major difference is that the resulting class instance is a much more complex structure than

a simple variable. An instance of the class provides a combination of data structures and operators on those data structures" (28:52).

The class constitutes a more powerful unit of reuse than simply reusing a procedure. "The benefit of reusing an artifact is related to the artifact's abstraction level. The higher the abstraction, the higher the potential payoff" (46:342). "The appeal of all this is the possibility that the software industry might obtain some of the benefits that the silicon chip brought to the hardware industry; the ability of a supplier to deliver a tightly encapsulated unit of functionality that is specialized for its intended function, yet independent of any particular application" (13:26).

"Class inheritance supports a style of programming called programming by difference, where the programmer defines a new class by picking a closely related class as its superclass and describing the differences between the old and the new classes" (26:23). The programmer immediately reuses the classes in the inheritance hierarchy above the new object being created.

Inheritance and composition extend the reusability of classes. "Related objects may be grouped together to form frameworks and toolkits" (46:343). "A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes" (26:22).

Through the support that the object-oriented paradigm provides for reuse, the paradigm has the potential to transform programming from a solitary cut-to-fit craft into an organizational enterprise like manufacturing (13:27). "This means letting consumers at every level of an organization solve their own software problems just as home owners solve their plumbing problems: by assembling their own solutions from a robust commercial market in off-the-shelf subcomponents, which are in turn supplied by multiple lower level echelons of producers" (13:27).

2.6.1.4 Promotes Maintainability and Extensibility The object-oriented model supports the principles of maintainability and extensibility

through encapsulation and inheritance. Through encapsulation the object-oriented paradigm allows programmers to build systems that are resilient to change.

According to Booch, the linkages between abstractions will change less than the abstractions themselves (3:11). The paradigm allows you to package volatility within problem-domain constructs (classes), thereby providing stability over changing requirements and similar systems (11:17). Maintenance is thus made easier because it becomes localized in specific spots — classes.

Encapsulation (as embodied by the concept of the class) also facilitates extensibility. "The object-oriented design process produces designs which facilitate the integration of individual pieces into complete designs. The narrow, clearly defined interface of a class supports integration with other software components. The narrow interface corresponds naturally to the observable behaviors of the real-world entity modeled by the class" (28:52).

Inheritance makes maintenance easier. "Inheritance mechanisms reduce the likelihood of human error because changes in one class are automatically propagated to all subordinate classes" (60:49). The alternative would be to modify all the methods everywhere they occurred.

Inheritance facilitates the extension of existing programs. "It allows extensions to be made to a class while leaving the original code intact. Thus, changes made by one programmer are less like to affect another" (26:23). "Class inheritance permits a new version of a program to be built without affecting the old" (60:48).

2.6.2 Drawbacks of the Object-Oriented Approach There are three acknowledged drawbacks to using the object-oriented approach: 1) performance considerations, 2) startup costs, 3) lack of direct support for constraints present in an object-oriented system.

2.6.2.1 Performance Booch discusses possible performance risks involved in using object-oriented languages (3:216-217). The first risk derives from late binding made necessary by polymorphic references. Booch indicates that polymorphic method invocation takes from 1.75 to 2.5 times as long as a statically determined method call.

A second source of overhead comes from the "layering" evident in object-oriented systems. Methods are generally small and tend to build on lower level methods. "This plethora of methods means that we end up with a glut of method invocations. Invoking a method at a high level of abstraction usually results in a cascade of method invocations; high-level methods usually invoke lower level ones, and so on" (3:216).

A third source of performance degradation comes from the dynamic allocation and destruction of objects. Dynamic allocation of objects costs more in computing resources than statically allocating an object. "For many kinds of systems, this property does not cause any real problems, but for time-critical applications, one cannot afford the cycles needed to complete a heap allocation" (3:217).

2.6.2.2 Startup Costs Booch and Coad (3, 11) both address the problems that are encountered when an organization moves to an object-oriented approach. "Using any such new technology requires the capitalization of software development tools" (3:217). "It is common to see organizations adopting object-oriented analysis and design if they are using a language like Ada or Smalltalk or if they view the transition from C to C++ as relatively minor. On the other hand, it is less common to see business-oriented data processing organizations adopting object-oriented analysis and design — simply because it is less obvious how it will work with COBOL" (11:156-157).

Another drawback is that the object-oriented paradigm requires a fundamental shift from the traditional thought processes involved in making software. "Using

object-oriented design for the first time will surely fail without the appropriate training. An object-based and object-oriented programming language is not "just another programming language" that can be learned in a three day course or by reading a book. It takes time to develop the proper mindset for object-oriented design, and this new way of thinking must be embraced by both developers and their managers alike" (3:217-218).

2.6.2.3 Constraints "A *constraint* is a numeric or geometric relationship between objects. They are described in terms of *visible* aspects of the objects in question, and they define a set of rules limiting the number of correct states of the set of objects. Constraints comprise two aspects: One aspect is the *declarative* aspect; the definition of the constraint. The second aspect is the *procedural* aspect, namely the actions taken when the constraint is violated" (29:27).

"An example of a constraint would be that two views of the same data remain consistent (for example, bar graph and pie chart views)" (19:25). Another would be that object1 must always be 5 pixels to the left of object2 (29:28).

The problem is that the object-oriented paradigm does not *directly* support constraint programming. Constraints must somehow be reflected in the attributes and methods of the objects in the system. "Currently, the combination with constraint programming remains one of the unsolved problems of object-oriented programming. On the one hand, we want to have the encapsulation and modular aspects advocated by the object-oriented approach. On the other hand, we want to have a more declarative approach that emphasizes more on *what* we want to solve, instead of *how* we should solve it" (29:28).

2.7 Successful Applications of the Object-Oriented Approach

While object-oriented programming has been around since the early 60's, the object-oriented approach to software development is relatively new. Coad and Your-

don advise that "you will have to decide for yourself" whether the object-oriented paradigm is sufficiently mature to attempt to use (11:156). Despite this relative immaturity, the object-oriented approach has successfully been applied to a wide range of software applications.

Object-oriented techniques have been used at the General Electric Research and Development Center (GE R&D) to develop compilers, graphics, user interfaces, databases, CAD systems, simulations, meta models, control systems, and even another object-oriented language (47:9). AT&T Bell Laboratories used C++ to construct a program debugger (9). The Apple Corporation used object-oriented techniques in making the MacApp object-oriented application framework in 1985 (60, 3). Both Keith Gorlen and Grady Booch have produced low level class libraries for general purpose use (21, 5).

Object-oriented techniques have been successfully used to implement various types of simulations. The Software Engineering Institute and ITT Research Institute have both used object-oriented practices in making flight simulators (30, 31, 36). The Mitre Corporation has produced a Large Scale Air Defense Simulation (62). The USAF is currently developing an object-oriented simulation environment for airbase logistics (41). The general applicability of the object-oriented paradigm to simulations has been advanced by various authors (15, 2, 48).

2.8 Summary

This chapter has introduced the current view of the object-oriented paradigm. The object-oriented paradigm represents a more intuitive way to program than using procedurally oriented techniques. The object-oriented approach is based upon the concept of an object: an abstraction that contains both data and the operations that modify the data. The implementation of the object is hidden. Objects in an object-oriented system model their counterparts in the "real world" of the problem domain.

Object-oriented programming has been around since the early 60's but the techniques for object-oriented analysis and design are much more recent. Techniques for object-oriented analysis, design and implementation all operate upon the same building blocks — the objects of the system. Thus, the object-oriented paradigm is seamless in that there is no translation of products from one phase of the lifecycle to the next.

The object-oriented paradigm offers a way to manage the complexity inherent in software systems. It produces systems that are more reusable, maintainable, and extendable than systems developed with procedurally oriented methods (60). The object-oriented strategy has been applied to a wide range of applications. Object-oriented techniques will provide the clarity and flexibility essential to the successful development of tomorrow's complex systems (60:11).

III. Methodology

3.1 Introduction

This chapter describes the steps taken in the phases of the formulation and implementation of the design for the flight simulator. The methodology presented is a synthesis of the various practices and suggestions that were described in chapter 2. The discussion begins with the analysis phase of the project and introduces the notation used. The next section covers the high level design techniques employed in continuing beyond the analysis. The last section of this chapter details the low level design (implementation) portion of the system.

Despite the fact that these phases are being presented as distinct and in sequence, in practice, this was certainly not the case. The development of the design was an iterative process. Once the initial analysis was done, the smaller parts were fully designed and implemented. During the course of the design and implementation, the nature of the relationships between objects that had not reached the design stage and those that did became much more evident. This, in turn, lead to changes in the analysis results.

The iterative nature of the process also applied to the activities performed during the separate phases of analysis, high level design and low level design. These steps were by no means performed in sequence nor were they done in isolation. Except where a specific sequence is indicated, they are best considered as activities that must be performed along the way to a complete product. It didn't matter when they were done or in what order they were accomplished, only that they were completed. They are not meant to be used as part of a "cookbook" approach.

One of the effects of this iteration was that the line between the phases became blurred. Parts of the design were fully implemented at the same time that other parts were still being more fully analyzed. Elements of all of the phases could probably be

found in any object at any time. The proportions of elements fitting into each phase simply shifted from analysis to high level design to low level design as the process progressed. The method can be summarized as follows:

I. Analysis: Construct a model of the "real world" system.

   A. Context Diagram: Define the problem and its boundaries.

   B. Identify Classes/Objects: Identify the major abstractions in the system with respect to the problem to be solved.

   C. Identify Structures: Find the relationship between the classes in the system.

   D. Identify Object Attributes: Define the characteristics of each object.

   E. Define Methods: Specify what functions each object needs and provides.

II. High Level Design: Transform the model gained in analysis into a form suitable for a computer.

   A. System and Structure Refinement: Map how objects can be implemented.

      1. Class Refinement: Examine each class from the analysis with respect to reusability concerns.

      2. Identifying Code to Reuse: Save development time by using existing code.

      3. Resolving Multiple Relationships: Specify how multiple relationships are represented in the system.

      4. Inheritance Structure Refinement: Factor out commonality between classes and standardize protocols.

   B. Concurrency: Identify the possible concurrency in the system.

   C. External Data Requirements: Handle external sources of data and persistent object storage.

III. Low Level Design/Implementation: Continue design activities into the coding of the system.

A. Object Representation: Decide how to implement the object: a class, data type or static object.

B. Implement Object Methods: Implement the methods with respect to the design strategy. Can add new attributes to help out.

C. Establish Object Visibility: Ensure necessary data paths exist and explicitly hide information within each object.

D. Identify Polymorphic Methods: Specify appropriate polymorphic methods.

3.2 Analysis

While the focus of this thesis is the design of the flight simulator, it was not possible to simply start with a design. The analysis had to be done first in order to have a starting point for the design. I was not trying to analyze the general problem domain of aircraft flight. I concentrated my efforts on analyzing an existing aircraft flight simulator. The main reasons for this were: 1) the problem domain of a flight simulator is more defined than the more general problem domain of aircraft flight, and 2) I had access to a variety of simulators. The analysis was based upon the things that I could see by using the simulator and was not concerned with the actual computer code used to run the simulation.

The main purpose of the analysis phase was to produce a model of the problem. The product of the analysis was a high level view of the classes and objects in the simulation. This high level view was then used to make an initial division of labor between the three individuals implementing the design. This allowed all three individuals to essentially work alone on their respective portions until the time came to integrate the pieces.

The initial analysis was perhaps the most important part of the process. The success of the initial analysis can be measured by the amount of interaction necessary between personnel who are assigned to work on supposedly separate objects (or object hierarchies). A low amount of necessary interaction indicates a good initial analysis. This was the case with this system. Development of the supposedly independent pieces did, in fact, proceed independently. If a high amount of communication becomes necessary, it is a signal that the pieces may not have been as distinct as first thought.

3.2.1 The Analysis Process The analysis process used was based mainly upon the methodology advocated by Peter Coad and Edward Yourdon (10). Without going into detail and comparing one author's method with the next, I chose Coad and Yourdon's method because: 1) it is widely applicable, 2) it covers the four components of analysis summarized in chapter 2, 3) it makes almost no assumptions about how the analysis will eventually be implemented (it assumes only inheritance), and 4) it includes a notation that can graphically depict the elements of the analysis and the design. The specific steps in the analysis process were:

1. Perform a context analysis.
2. Identify the classes/objects.
3. Identify structures — inheritance and whole-part relationships.
4. Identify object attributes.
5. Define methods (includes interobject communication).

The last four steps are included in Coad and Yourdon's method, the first step is not.

The purpose of the first step is to define the problem prior to analyzing it. The context analysis provided a general definition of the problem and what the boundaries of the system were to be (57). The context analysis consisted of four elements:

1. The problem statement: a one paragraph description of the system under consideration.
2. A concept map: a graphical depiction of the system within its immediate environment.
3. An event list: a list that delineates the outside stimuli that the system must react to.
4. A narrative constraint list: defines the economic, technical and legal constraints imposed upon the development of the system.

The context analysis is included as appendix B.

Once the problem was defined, the analysis began. The second step in the process was to identify the classes/objects in the system. This was accomplished by using the tips for finding classes/objects listed in section 2.5.1.1.

The best sources of information used to identify the classes/objects of the new flight simulator were existing flight simulators. I was able to observe and fly several different flight simulators ranging from a multi-million dollar model to several different pc-based simulations. The most useful simulations were the "Flight" and "Dog" flight simulators made by Silicon Graphics Corporation. "Flight" and "Dog" were particularly useful because they ran on the workstations on which we were going to implement the new simulator.

The existing flight simulators were used to identify classes/objects mainly through the entities present in the simulator, information that was tracked, and the external devices that the simulators interacted with. The most significant realization was that the main components of a simulator were what was visible to the user — in essence, the simulator consisted mainly of moving pictures. The information that was tracked pointed to candidate objects/attributes and it was a natural progression from identifying external devices to making them classes/objects.

"Flight" and "Dog" also served another purpose. With no detailed specification nor specific guidance pertaining to the capabilities of the new system, the two SGI programs were used as examples of what the new simulator should provide. Thus, the SGI programs served as the main source of the requirements for the new simulator.

The third step in the analysis process was to determine the structure inherent in the system. The strategy was to decide which objects were related and then classify the relationship as a composition (or "uses") or as an inheritance (or "kind of") relationship. The simple test was to ask whether object "A" was a kind of object "B" or whether "B" simply used "A".

The problem was analyzed from the knowledge that was available from the domain in question. While some relationships were easy to classify, there were no hard and fast rules to apply that would exactly specify the type of relationship between two or more objects.

The question of multiple inheritance slightly complicated matters. As it turned out in the analysis phase, there were no objects that appeared to require a multiple inheritance relationship. We did eventually implement one object using a multiple inheritance relationship but this was only to make the object easier to use. Grady Booch's rule of thumb about "the sum of the parts" (3:116) worked well in determining if a relationship was truly one that required multiple inheritance.

The next step was to specify the attributes of the objects. This step required the most knowledge of the problem domain and the requirements of the system. Answering questions from the perspective of an object such as "How am I described?", "What do I need to know?" or "What state information do I need to remember over time?" (10:121) are impossible if one knows nothing about the object or what is required of it in the context of the system. The only recourse is to learn the necessary information.

The basic strategy for determining attributes was to use an initial cut at the overall composition of the system and attack the more familiar objects first. The idea was to specify the objects that were the most understood at the outset. The benefit of this was that the system was more defined when it came time to attack the more complex objects later in the analysis process. As an example, the hierarchy of window objects was fully specified before the objects that contained the flight dynamics of the simulator. This made the task of specifying the less understood "flying" objects easier later on.

The last step in the analysis process was to specify the methods provided and required by each object. The first action was to specify the "selector" and "modifier" functions for each attribute of each object. These were specified first because they are the simplest methods of a class. A "selector" function simply returns the value of an attribute while a "modifier" allows the user of the object to set an attribute (3). These are the "Get" and "Set" methods that are present in the current design (refer to appendix C).

Once the primitive selector and modifier services were specified, the next step was to add the "characteristic" services to the object. I relied upon the observation that objects will not normally exist simply to allow users to get and set their attributes one at time. There was always a reason why particular attributes were grouped together the way they were. The characteristic functions defined the real purpose of the object and they depended upon what the object existed to do.

The last step in the specification of methods was to define what methods were needed from other objects — the interobject communication. The interobject communication was determined by the needs of the characteristic functions. The case encountered most often was where one object was a component of another and the composite needed the selector or modifier functions of the component. Every attempt was made to reduce or eliminate the need to get information from an object that was not a component or part of the inheritance structure of the calling object.

3.2.2 The Notation Coad and Yourdon's notation for graphically depicting the design was the basis for the notation used in this thesis (see figure 2.2). While I have added or modified this notation in certain areas, the essential strategy is the same. I have added to the notation to give a better view of the actual character, requirements and limitations of the classes/objects in the design.

There are two basic types of notation: the system notation and the object notation (see figures 3.1 and 3.2). The system notation depicts groups of classes/objects and the relationship between them. The object notation covers one class of objects in detail.

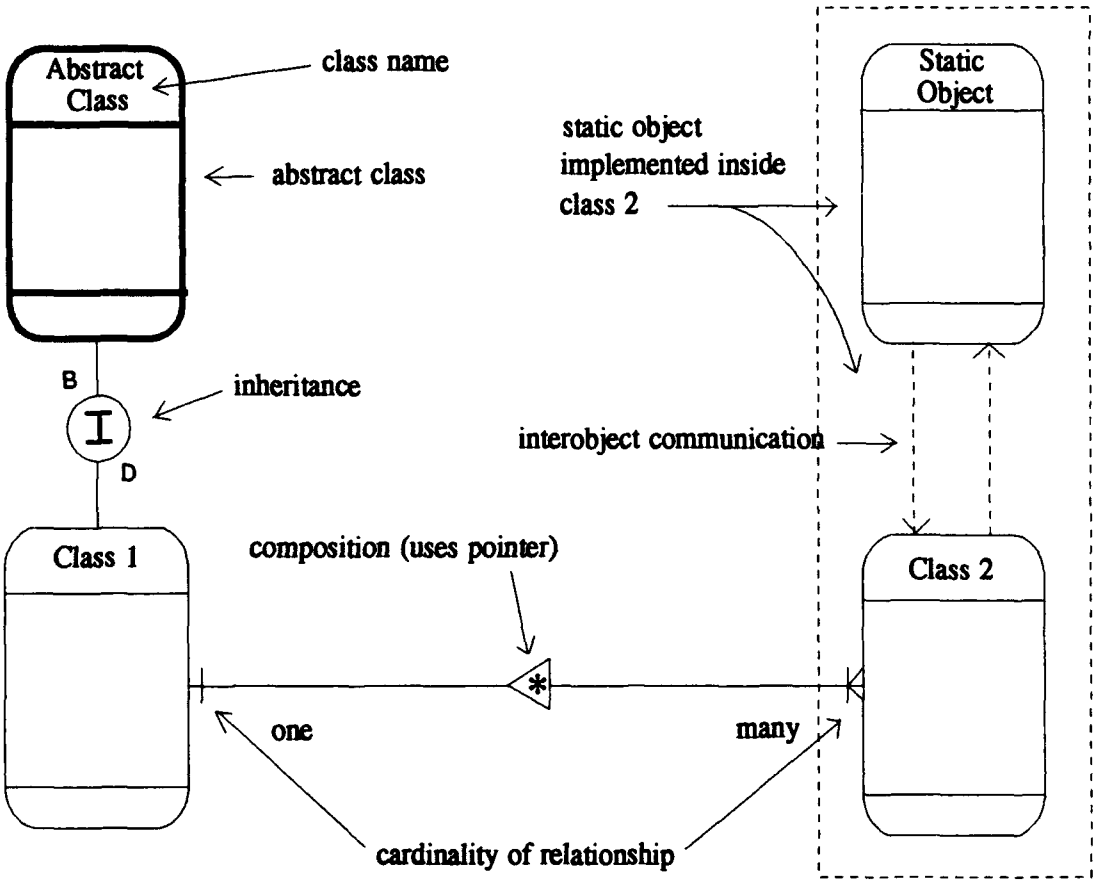


Figure 3.1. System Notation

The purpose of the system notation is to give a high level view of the layout of the system. In the system notation, the inheritance and composition relationships are depicted as well as any needs for services from objects outside the hierarchy of a particular object. Only the name of the class/object is listed in each rounded box. The number of classes/objects shown on any diagram should not be more than can be understood at one time. (Coad and Yourdon recommend 7-9).

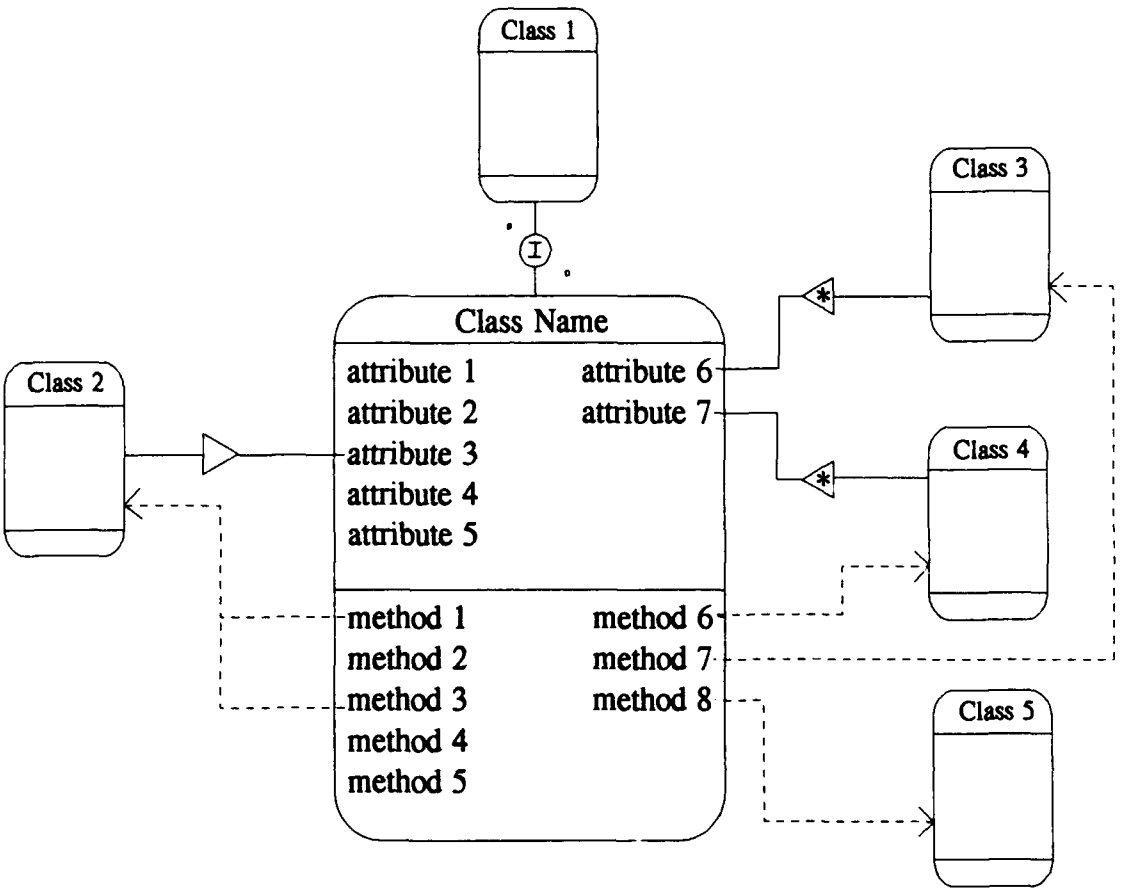


Figure 3.2. Object Notation

The object notation is a detailed view of one object at a time. The object notation uses the same symbols as the system notation and adds more to it. The rounded box for the object being shown is divided in three areas. The name of the class/object is located in the top area. The attributes of the object are located in the middle section and the services are listed in the bottom. Arrows are drawn from services to system level depictions of other objects if services are needed from the other object. All inheritance and composition involving the target object are depicted as well. The purpose of the object notation is to depict how the object is coupled to other objects. The notation that I have devised contains design and C++ specific parts:

1. The inheritance symbol has been changed from a half circle to a full circle with an "I" in the middle. A small "D" is placed next to the line going to the derived class and a small "B" is placed on the line to the base class.
2. The composition triangle now contains an asterisk if the component is actually a pointer to an object of that class.
3. Pure virtual (abstract) classes are drawn with thicker lines than normal classes.
4. Rounded boxes depicting classes implemented as static data members and methods are shown inside a dashed box with the class within which the static members were implemented.

With these additions, it became possible to get a much more detailed view of the design. However, the design and implementation dependent notations should not be used in the analysis phase. The only portions that should be used in analysis are the inheritance, composition, class/object, and line of communication graphics. All of the other portions are design or implementation dependent and should not be added until the design and implementation phases.

3.3 Design

With the initial analysis complete, the design of the system began. As Meyer points out, object-oriented design is primarily a bottom up approach (34:325). The reason for this is that it is not possible to construct the composite higher level objects of the system without first making the simpler objects of which they are composed. This was certainly the case with the flight simulator. The simpler (least complex or well understood) objects were designed and implemented first.

While it was certainly true that the overall process was primarily bottom up, we were able to do some top down design as well. As simpler objects were created, they were used in higher level objects. In this manner, we were able to incrementally complete the higher level objects as more primitive ones became available for use.

The bottom up nature of object-oriented development turned out to be a major factor in managing the complexity of the system as a whole. Once the initial analysis was done, the primitive objects were the first to be implemented. This had two effects: 1) the completed objects clarified the design, 2) once constructed, the complexity within the primitive objects could then be ignored.

3.3.1 Good Design/Designing For Reusability There were many suggestions offered for making a good design in chapter 2. The overriding concerns that motivated the majority of the suggestions were to reduce coupling, increase cohesion, and to enhance encapsulation of the details of an object. These three qualities lead to classes that are reusable, extendable and maintainable. The overall idea is to produce a class that users can simply pick up and immediately use in other applications (like Cox' software ICs).

The main goal in designing and implementing each class of this design was reusability (maintainability and extensibility were derived from designing for reusability). While I tried to use all of the suggestions in chapter 2, I ended up with additional guidance to use when designing classes.

The main idea was to look at the class from the perspective of a potential user of the class. All classes were constructed from the standpoint that they may be used in other, possibly unrelated applications. The most effective way of accomplishing this was to look at the class in isolation from the rest of the system being built.

The classes were designed to be easy to use. Using the class means two things: the use of instances of the class and using the class as a base class from which to derive more specialized classes. Making the class easy to use means something different for each type of usage.

Making the class easy to use for a designer who plans to use the instances of a class is centered totally upon the methods that are offered by the class. When used in this manner, the class can be viewed as a "black box". Information hiding is the

overriding principle in making objects of the class easy to use. The designer intent upon reusing the class is not concerned with the internal details of the class, only with what can be done with/to objects of the class.

The question becomes what kind of functionality to provide through the methods of the class. The design concepts summarized in chapter two that are particularly applicable in this situation are sufficiency, completeness, primitiveness, clarity of design and simplicity. Allthough it was not mentioned in chapter 2, the concept of "robustness" also applies. In short, the designer should provide all useful methods that the object could provide using *only* the state information contained within it.

The concept of "robustness" means that the object should be built to be tolerant of errors. The object should have the capability to somehow inform the user that a method has failed. The object should not crash the program due to an error. The error should be passed back to the user so that the user can decide what to do. In addition, every effort should be made to ensure that users are prevented from ever causing an error condition in the first place.

Making classes easy to reuse as base classes is a much tougher exercise than simply designing a class where the objects of the class are to be the focus of reuse. The main problem is that reuse in this context is a "white box" exercise. The designer who intends to reuse the class as a base class must necessarily know the details of the implementation of the potential base class — an idea contrary to the principle of encapsulation (53).

The problem with designing reusable base classes does not lie with child class accessibility to attributes of parent classes. A measure of a good inheritance structure is when each more specialized class can truly be considered as possessing all the attributes of the parent class (and all its ancestors). In general, there should not be a situation in which a designer should hide an attribute in a superclass from lower level classes — a designer should not seek to "uninherit" attributes.

Seeking to "uninherit" attributes or methods is a sign of poor factorization. If it becomes necessary to hide attributes, then the question becomes "was child class B really "a kind of" base class A?". The inheritance hierarchy in question should be rearranged and/or split up so that only the methods or attributes that are needed are actually inherited. The benefit of this is a more understandable structure.

The main problem with designing reusable base classes lies with using the methods of the parent class and its ancestor classes in implementing the new child class. At this level, the concern is with reuse of methods. Because there is no encapsulation, this becomes very similar to reusing procedures in a procedurally based program. The attributes of the class hierarchy can be considered as global variables with respect to the methods.

Cohesiveness became the primary concern when building methods of classes intended for use as base classes. While this resulted in a few more methods in such classes, the classes were more reusable because the methods could be used more effectively. Making smaller and more cohesive methods reduced the need for duplicating code between parent and child classes.

The methods of base classes were also made to be self-contained. This made using them simpler in that a user would have less to worry about when trying to combine simpler methods to construct a higher level function. Making the methods as self-contained as possible extended the reusability of the class itself.

3.3.2 High Level Design The purpose of the high level design phase was to make decisions concerning the form that objects would take in the implementation. The purpose of this phase was to bring the analysis into the realm of design. The result was to put the products of the analysis into a form suitable for implementation on a computer. The focus turned from the "what" of analysis to the "how" of design.

The first action of the high level design process was to settle upon a "design strategy" for the construction of the design. This corresponded to step 8 of Rum-

baugh's "system design": "Set trade-off priorities — make a decision as to what gets priority during the development. Decide between such factors as speed, memory available, portability, functionality, cost and time available" (47:199-211). The design strategy served as the overriding concern when making design decisions.

We made the decision to give speed of the running simulation highest priority. Real time computer graphics such as those that we were planning to display in the flight simulator place enormous demands upon the processing power of the computer (49:48). Given this and the fact that we had a 15 frame per second requirement for the system, the natural choice was to optimize the design for speed.

The potential memory requirements were the secondary concern. We realized that the storage requirements for the system could be a potential problem. However, the general attitude at the start of the effort was that we would let the virtual memory manager handle these problems. While we made some design decisions with memory in mind, speed considerations always took precedence over storage concerns.

The method that I used to make this particular design is a composite of the practices advocated in chapter 2 with some additions. The design practices that were used can be distilled into three main subject areas: 1) refinement of the system and its structure, 2) concurrency, and 3) external data requirements.

3.3.2.1 System and Structure Refinement The main activities of the refinement phase were: 1) Adding additional methods and attributes to existing classes to make them more reusable, 2) identification of previously existing software that could be used in this design, 3) making decisions on how to represent multiple relationships in the analysis, and 4) further modification to inheritance hierarchies. The objective of this phase was to map how the object(s) could be implemented. The order in which these steps are applied was not important. The idea was to make sure that they had all been accomplished.

3.3.2.1.1 Class Refinement The purpose of the class refinement activity was to examine each class in the analysis with respect to the reusability factors mentioned previously. The analysis yielded a model of the real world system under consideration. Therefore, some of the classes in the analysis lacked methods and attributes to make them more reusable in other domains. The methods and attributes that were needed to "round out" the classes were added in this step.

3.3.2.1.2 Identify Existing Code to Reuse The purpose of this step was to look for existing code that could be reused in the design in hopes that this would reduce the overall effort required. The best place to start looking would have been other object-oriented systems that were similar to our application. Unfortunately, there were none. The code that we reused came from the UNIX system library, the SGI graphics library, and the SGI flight/dog program.

The motivation to reuse code was driven by the application and the available routines. This phase of the design was often done in concert with other phases in the design process, such as handling external data requirements. For example, the character of the system was such that certain entities were required to perform input/output functions given the available system/library routines. These entities were based upon the existing code.

The objective at this point in the high level design phase was to specify in general terms what these procedurally based entities were and what they would do. The specification included what the attributes were and what methods would be offered by the new class. The implementation of these new methods would take place during low level design.

Once the sources of code were identified, the next step was to fold the available pieces into an object-oriented framework. If we had another object-oriented system, we would have simply taken whole classes out to reuse. Given that what we had was essentially a collection of procedures, the procedures became methods

of objects outright or were going to be called from within methods. The data that the procedures used became attributes of the new objects.

Rumbaugh (et al) calls a class that is based upon library or system routines a "wrapper". Wrappers were constructed to make using the library routines easier, to extend the functionality of the routines and to localize where the system routines were called from. This provided the benefit that the system calls were encapsulated and their usage became localized in one place. If the system calls ever changed, the design would only be impacted in one area.

This step was very close to being low level design. This is due to the fact that the tools of this step are implementations. The goal of this step was to organize the available routines into an object-oriented framework. The existing procedures and the data that they used essentially constituted a skeleton for a class that was filled out with additional methods (how to fold existing code into an object-oriented framework is the subject of chapter 6).

Not all system routines needed to be folded into classes. Deciding whether to encapsulate system routines was dependent upon the number of functions that referred to the same data. The key was to look for a group of data elements that were used by multiple system calls in order to provide a different level of functionality than that provided by simply returning or changing the data elements. This was an indication of a candidate class. I mainly looked for characteristic functions given a set of data that multiple routines used.

3.3.2.1.3 Designing Multiple Relationships Part of the analysis revolved around defining the multiplicity of the relationships between classes/objects that participated in a composite relationship. This step in the high level design process involves deciding how many-to-one and many-to-many relationships will be implemented. These decisions were made at a high level of abstraction and were not implementation dependent.

The purpose of this step was to identify the strategy that would be used to implement these multiple relationships. All of these relationships entailed some type of list structure. Choosing the appropriate collection mechanism was the main activity in this step.

Choosing the exact type of list was done with a number of factors in mind. First and foremost was the fact that we were designing this system to be as fast as possible. With speed as the overriding interest, the other factors that determined which type of list to use were:

1. Frequency of access: How often would the list be accessed?
2. Type of access: How would the list be used? Would this require an ordered list? Was access to specific objects needed? Was the list simply going to be iterated from start to finish?
3. Frequency of additions/deletions: How often would the list be changed? Would the user always add and never delete from the list? How many objects did we expect there would be in the list at any one time?
4. Nature of the relationship between objects in the list: Was it enough that the objects were in the list (perhaps in some particular order) or was there a requirement for a more sophisticated relationship between the objects in the list (eg. one that would require a generalized list)?

Once the type of list was determined, the next step was to add the necessary classes and modifications to existing classes to incorporate the selected list into the design. When this step was finished, there were no multiple relationships that led to classes that didn't have some type of list associated with them.

3.3.2.1.4 Inheritance Structure Refinement Modification of the inheritance structure in the program involved organizing the design to take advantage of similarities between classes, standardizing protocols and identifying abstract

classes. All three of these activities were very closely related. The goal of inheritance refinement was to further organize the design and to take advantage of commonality between classes.

The first step was to try to factor commonality from the classes in the inheritance structure. New classes were created where they could be used by more than one subclass or if they constituted a usable abstraction in themselves. This was done with one warning in mind: "Do not create levels of specialization simply for the sake of doing it" (11:143). The reason for this "second look" at inheritance was to look at the design in light of the fact that new classes were added since the analysis.

Once commonality had been identified, standardization of protocols came next. Standardization of protocols was accomplished by defining additional "root" superclasses that contained the description of attributes and methods that should be provided by all subclasses derived from them. As Korson suggests, the new root class of an inheritance structure should be an abstract model of the target concept (28:54). Standardization of protocols aids reusability by defining a default interface for all the different variations of the root classes. The standard protocol and attributes also serve as a starting point for new derived classes.

The last step in inheritance refinement was to identify abstract classes. In the course of factoring the inheritance structures, there were classes that served only as parts of others. The root classes of the inheritance structures fit into this category. The abstract classes existed only to provide pieces for derived classes. Instantiating objects of an abstract class would make no sense because they lack the necessary attributes and/or methods to be usable. Identifying classes used to instantiate objects served to define the usable "leaves" of the inheritance structure tree.

While it was not part of this particular application, accommodating the supported level of inheritance would also belong in this step. This would correspond to step four in Coad and Yourdon's Problem Domain Component (11:39-48). If the

target language does not support inheritance, then the design would be altered in this step to take the supported level of inheritance into account.

3.3.2.2 Concurrency This step revolved around identifying which parts of the design would be running at the same time. This was essential in determining the type of control present in the program. The operating system, the implementation language and the computer all determined what type of concurrency would be possible.

Coad and Yourdon's guidelines for building their Task Management Component all applied here (11:73-76). However, their guidelines had to be carried out in light of the capabilities of the hardware and software upon which the system was implemented. It would not have made any sense to design a concurrent system if concurrency wasn't supported.

Concurrency was not added simply because it was possible to implement. The requirements of the application drove the decisions to include concurrency. In general, it may be unavoidable to include concurrency in a system in some cases and purely optional in others. The decision to include concurrency was made with the extra processing requirements and the limitations on using concurrency presented by the hardware and software in mind. The main limitation in our case was the fact that it was much slower to communicate between multiple processes versus implementing the system within one process.

3.3.2.3 External Data Requirements The motivation for this step was to incorporate external data sources/sinks into the design. This involved identifying the sources/sinks of the data and defining how the data was to get from the external device to/from the computer. This step also involved specifying those classes responsible for storing and retrieving (object) data and defining the required data persistence strategies. This goal of this step was to recognize and accommodate the external input/output sources of the system.

The products of the analysis phase included the descriptions of classes that encapsulated the functions and data of the external stimuli. Given this description, the next step was to define low level classes to perform input/output with the device(s) and the computer. This definition included how these "bridge" classes were related to the external device classes. Since these lower level classes were normally implemented using procedure calls provided by the operating system, this phase often overlapped with defining how to best use system and library routines (see section 3.3.2.1.2).

The second part of this activity was to identify where the system would store and retrieve data about itself. This was motivated by the need to store object data from one invocation of the program to the next. The classes responsible for achieving persistence were identified and modified for the task.

3.3.3 Low Level Design/Implementation The purpose of the low level design phase was to concentrate on implementing the design given the strengths and weaknesses of the language used. I have used the term "low level design" as opposed to just "implementation" to emphasize the point that the design process does not stop when coding begins. Implementing the design is not a simple matter of just coding up the system.

The method used for low level design was derived mainly from the suggestions given by Rumbaugh (et al) for doing object design. The activities of low level design included: 1) deciding upon object representation, 2) implementing object methods, 3) establishing object visibility, and 4) identifying polymorphic methods. All of these activities are closely related.

3.3.3.1 Object Representation This is derived from step 7 of Rumbaugh's object design strategy (47:228-249): "Determine object representation — decide whether to use primitive types (integer, string, real, etc.) or implementation as an object. SSAN is a good example. Do you make it an object or simply im-

plement it as a string of 9 characters?" This step depended upon how the C++ language could be used. For example, C++ allows the user to use any valid "C" construct in addition to the mechanisms provided for object-oriented programming. It was not required to make everything an object (unlike the Smalltalk language which requires that everything be an object).

The method used for deciding how to implement a class was to gauge the functionality of the candidate class and implement it accordingly. The best indicators of functionality were the characteristic methods of the class, or rather, the lack of characteristic methods. If the candidate class contained nothing but selector and modifier methods on all attributes ("Get" and "Set" methods) and did not serve as a base class, then it was implemented as a data type. There was no reason to complicate using these simple abstractions by implementing them as classes when it was not required to encapsulate anything about them.

Another issue was where to put the definitions of these types. The simple solution was to encapsulate the type definition within the class that used the type. If other classes needed the same data type (as was the case with the three dimensional "Point" struct), then the definitions were placed in a global file. Types that were used internally in the class were hidden from users (other languages may not provide this capability). However, some types were made public to users of the class to make the class easier to use and/or more robust.

3.3.3.2 Implementing Object Methods Step 2 of Rumbaugh's object design method applies here (47:228-249): "Design algorithms to implement operations — choose algorithms that minimize the cost of implementing them. You may define new classes and operations as necessary." I also added attributes to classes when the implementations of objects made them necessary. Methods were also implemented with the design strategy in mind. In this case, it was to build for speed.

Cohesion was the foremost principle used in implementing methods. In some

cases, this involved splitting methods into smaller, more cohesive methods that existed only for use by the object to implement more complicated methods. These smaller methods were hidden from potential users because they didn't present enough functionality in and of themselves.

New attributes were also added to the class during low level design. The main reason for adding more attributes was to facilitate the execution of the methods. Some attributes were added to classes because their values were calculated frequently. The "airspeed" attribute of the User Aircraft class and the various "NTSC" attributes of the Text Item and Window classes are examples of this. Some attributes were added because of how certain methods were implemented. The "tty" attribute of the Port class is an example of this. The "tty" attribute was required by the UNIX system calls that handled RS232 ports.

Designing for reusability played a major role in this step. This meant keeping the number of parameters small and the names of the methods as descriptive as possible. Designing for reusability also entailed simplifying the parameters that were used. C++ allows a programmer to define enumerated types. I used enumerated types to both "spell out" a user's options to simplify interfaces and to make using the class more robust given the lack of a capability in C++ to define subtypes.

3.3.3.3 Establish Object Visibility This corresponds roughly to step 3 of Rumbaugh's object design method (47:228-249): "Optimize access paths to data — may restructure class organizations to optimize access, add attributes that store frequently calculated values or you may rearrange execution order for efficiency." There were two aspects to establishing object visibility: 1) ensuring that a way existed for objects to communicate and 2) defining exactly what was to be hidden in the object with respect to other objects in the system.

A C++ programmer has five ways to make one object visible to another: 1) inheritance, 2) make one object an attribute of the other, 3) make a pointer to one

object an attribute of the other, 4) implement one object as static class information and include the header file that defines the class or 5) pass the needed object as a parameter to all methods that require the object. These fairly limited ways of establishing visibility caused the design to change to reflect the manner in which visibility was established.

Once the necessary visibility had been established, the other side of the coin was considered — purposely hiding information. Different languages offer ways to explicitly declare the type of access allowed to a class. The C++ language offers a number of ways to establish fine-grained control of visibility to a given class (a discussion of how to exercise this control is included in appendix E).

As a general rule, all data members of a given class were hidden from all classes except those classes that inherited from the given class. Child classes were given full access to the data members of parent classes. No outside object has the capability to modify a data member of another class outside of the methods offered by the class that owns the data member. Granting unlimited access to data members would have directly violated the principle of encapsulation.

The main question about what to hide concerned the methods of the classes. Encapsulation was the main concern in this aspect of object visibility. A language that offered some way of specifically designating which classes could use each method of another class would have been ideal. This would have enabled us to hide even more information from classes that didn't have to see it. While C++ does not provide this ideal form of specification, it did offer a lot of control over visibility.

The methods introduced in the implementation of methods phase were all hidden from outside users. These methods were used only by the class to implement higher level methods within the class. No other class needed to know of their existence. Most all of the remaining methods were made available to users.

There were instances in which there were methods that some classes needed

that were best hidden from others. The C++ language provides a capability to grant unlimited access to a class through the "friend" keyword. This unlimited access was allowed in light of the benefits gained from hiding items from other classes.

3.3.3.4 Identifying Polymorphic Methods The concept of polymorphism was handled in low level design because it is primarily an issue of implementation. The use of polymorphic methods is simply another way in which to implement a method. The reason that this discussion was not included in section 3.3.3.1 was because the C++ language requires that polymorphic methods be specially designated within an inheritance hierarchy.

The best candidates for polymorphic methods were those methods that were common to all classes within an inheritance structure. This highlighted another motivation behind using abstract classes to act as "roots" to inheritance trees. After standardizing the protocols of lower level classes during high level design, the root classes could then facilitate polymorphic methods for all classes in the inheritance structure. With standard methods in place, it was a matter of choosing which ones would be polymorphic given how methods were to be implemented.

3.4 Conclusion

The analysis, high level design and low level design methods that I used were a synthesis of practices outlined in chapter 2 and of personal experience gained during this project. The analysis phase served to define the problem, define its boundaries, and to define the system in object-oriented terms. The initial analysis allowed us to split the project up into different parts such that little or no interaction was required with each other during the design of each portion. The analysis was based mainly upon Coad and Yourdon's method of OOA.

With OOA done, the high level design transformed the system into a representation suitable for implementation on a computer. All parts of the design were

constructed with the design strategy (to implement for speed) and good design practices in mind. The different parts of high level design included refinement of the object structure, determining concurrency in the system and recognizing the external sources of data.

The low level design concentrated upon the implementation of the system in C++. During this phase, object representation, implementation of methods, establishing object visibility and specifying polymorphism were the main activities. The low level design was the main source of "private" attributes and methods added to classes.

The analysis, design and implementation phases all went on at the same time. I was not constrained to complete one phase on the whole system and then move to the next. Parts of the design were still in the analysis phase while others had been completely implemented.

Overall, the system was constructed in an incremental, bottom up manner. Pieces were plugged into higher level objects as they were created in order to approach the system from the top down as well. By designing mainly from the bottom up, the system as a whole became more defined as more low level objects were done. This aided the design of higher level objects that had yet to be implemented. Aspects of analysis, high level design and low level design were all present in the system at the same time. Just as Grady Booch said, OOD was certainly an incremental, iterative process.

IV. Design Highlights

4.1 Introduction

The purpose of this chapter is to show how the design methodology presented in chapter 3 was actually applied. This chapter consists of a series of examples used to illustrate how the design evolved from one phase of the object-oriented design process to the next. The examples were chosen so that all the aspects of the methodology discussed in chapter 3 would be covered. The strategy for the analysis will be presented followed by specific examples of how certain classes were taken from analysis to design and through implementation.

Object-oriented design is an iterative process. The iterative nature of the design process complicated the explanation of the evolution of the classes. There were two options for the presentation of the explanation: 1) explain the actions taken to construct the design in the order in which they actually occurred or 2) arrange the actions by the design activity to which they belonged and order the discussion in the same order in which the activities were presented in chapter 3. I decided to used a hybrid of the two approaches.

The explanation of the actions taken to construct the design will be presented in the general order in which the actions occurred. The benefit of this approach is that it will best illustrate how the process actually unfolded. Within this organization, the actions will be categorized with respect to the subject areas discussed in chapter 3. The objectives to produce reusable code and to construct a good design permeated all phases of the process.

4.2 Analysis

As noted in chapter 3, I did not take a sequential approach to OOA. I did not identify all of the classes/objects, then affix some structural relationships between

them, then specify attributes, then finish by specifying the methods and which class needed them. My objective was to end up with an analysis that contained the necessary elements discussed in chapter 3. The strategy I took was to take a first cut at the all of the elements as I went and then go back and enhance the analysis where it needed more specification.

When doing the analysis one cannot consider the four elements of the OOA in isolation from one another. Often times, one element may be the motivation behind another element. For example, an inheritance relationship may be primarily driven by the fact that a child class must offer a slightly different or additional type of service.

The existing SGI Flight and Dog programs were used as the target(s) of the object-oriented analysis. They were the prime sources of the requirements for the system. The analysis focused upon the objects contained within the simulator and the external devices that were available for use.

A context analysis was done prior to beginning the OOA. The context analysis of the flight simulator is included as appendix B. The context analysis served to define the boundaries of the problem. Besides providing a general description of the problem, a benefit of the context analysis was the identification of the external factors that affected the system.

The following description of how the initial analysis was conceived is meant to convey a general sense of what motivated the initial view of the system. It is not meant to convey a detailed view of the analysis process. This is mainly because the initial analysis was not done in a detailed manner. It's purpose was to provide a framework that would enable more detailed analysis later in the process.

4.2.1 Initial Analysis The most important realization concerning the system was that the system essentially consisted of moving pictures. This was an application where "the picture is the thing" (18:293).

Thus, the analysis began from that standpoint and expanded from there. The "Graphical Object" class was the reflection of this line of thought in the analysis.

Starting with the models present in the system, it was apparent from using the Flight and Dog programs that there were two types: those that moved and those that did not. This was the justification for the "Dynamic Object" class which was inherited from the Graphical Object. The Dynamic Object class represented those models that had the capability to change position and/or orientation in some specified manner.

The Dynamic Object class would serve as the base class for all moving objects in the system. This included the specialized types of moving objects in the system: the Aircraft and the different types of ordnance. Further specialization of this inheritance tree produced two different classifications of ordnance and aircraft: those controlled by the simulator and those controlled over the network.

Once the basic pieces had been identified, attention turned to the object that would be responsible for characteristics of the imaginary world as a whole. This led to the "Virtual World" class. The main purpose of the class was to provide a way to manage the models and view the simulator world. The attributes of the Virtual World would include such factors as the time of day and weather conditions.

The identification of the Virtual World class led to the identification of the "Viewpoint" class. There could only be one view of the world used at a time. The Viewpoint would contain all the necessary attributes and services to specify and change the user's view of the virtual world.

With the Virtual World class in place, there was a need for an object that would act as the controller for the whole simulation. This object would coordinate and control all lower level objects in order to run the actual simulation. This produced the Flight Simulator class. Attributes in the Flight Simulator class included simulator time and an object of the Virtual World class. The main method of the Flight

Simulator class was the "run simulation" method.

This rough description constituted the objects that were internal to the system. The focus then turned to the objects that were external to the system. These included the "User Screens", "Joystick", "Polhemus", "Keyboard", "Mouse", and the "Network" classes. Each of these classes would encapsulate the attributes and methods needed to interact with external entities (the retrieval of the model definitions from the computer would be a method inside the Graphical Object). Lines of communication or composite relationships were added between the external classes and the ones internal to the system in order to show what the external classes would affect.

Figure 4.1 depicts the initial analysis as it stood on 10 Mar 91. Some aspects of it need further explanation. The lower left corner of the figure represents the fact that the F15 (an example name for a specific class of aircraft) would consist of attributes that would be affected by user inputs. The "IO Control" class was an attempt at trying to generalize input dependent classes under one root class.

Once the initial analysis was finished, we decided which parts of the analysis were to be given to each of the project members to design and implement. Captain John Brunderman was assigned the design and implementation of the Graphical Object class. Captain Robert Olson was given the task of designing and implementing the Viewpoint and Virtual World classes. The rest of the design and implementation was given to me.

Even though we did not realize it at the time, this division of labor corresponded to the Model-View-Controller concept (MVC) which is a part of the Smalltalk-80 programming environment for developing user interfaces (29:20). "The MVC-triad separates the three components — functionality, input, and output" (29:20). The Model represents the functionality in the system, the View contains the methods to make the model visible and the Controller reacts to user inputs and activates the methods of the model.

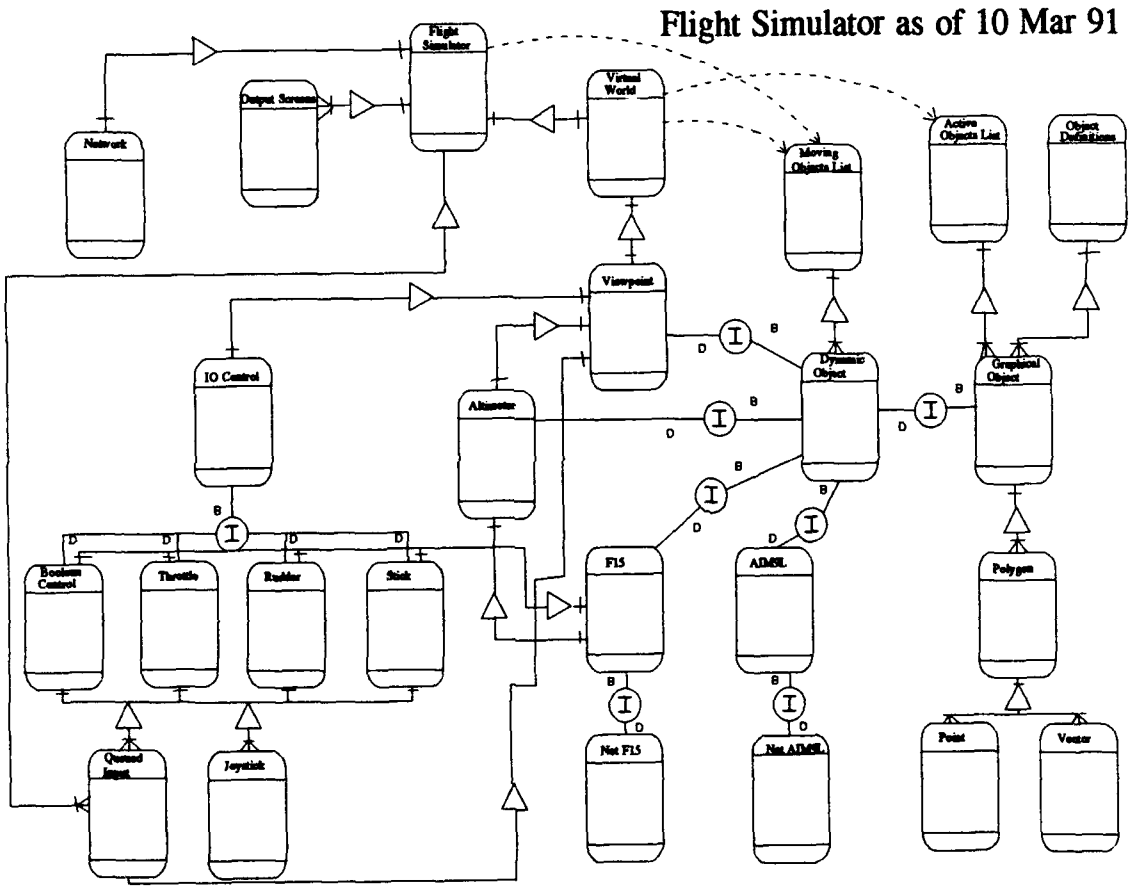


Figure 4.1. Flight Simulator Initial Analysis

Captain Brunderman and Captain Olson were responsible for the View portion of the triad. Captain Olson was responsible for the Viewpoint and Virtual World classes. These two classes would be responsible for how the world was viewed. Captain Brunderman implemented the Graphical Object class. This class was responsible for defining and rendering the individual entities in the system. Taken together, these classes constituted the View.

The Model and Controller portions remained. The Controller portion was made up of the external input device classes. These allowed a user to interact with the system. The Model portion consisted of the methods used to move the geometric models encapsulated in the Graphical Object class.

In retrospect, this initial analysis was wrong in some places and correct in others. It's main purpose was to provide an initial system to work with and it fulfilled this objective. Further discussions will highlight where, when and why this initial analysis changed character.

With the initial analysis done, we were able to consider the lower level classes in the analysis in detail. This marked the point where we began to shift pieces of the analysis into different phases. The lowest level classes were analyzed in more detail, designed and then implemented. The intent was that these implemented lower level objects would serve as building blocks for the rest of the system and would more fully define the remaining parts of the system.

4.3 Low Level Inputs — The Joystick and RS232 Port Classes

The classes necessary for handling the CH Products Microstick joystick were among the first to be implemented. The first order of business was to perform the analysis of the joystick class in more detail. The detailed analysis phase was followed by high level design and then an implementation. However, additional requirements for the joystick were identified after the first version of the Joystick class was put into use. The new requirements forced a second iteration of the high level design

and implementation phases for all classes that were related to making the joysticks work. Once this second version was done, even more capabilities were added to the Joystick class after it was integrated into higher level classes.

4.3.1 Detailed Analysis Given the initial analysis, the Joystick class was not derived from any other class nor did it require any methods from any other class. This indicated that the Joystick class could be implemented in isolation from the other classes in the initial analysis.

The main sources of information for the analysis were the user's manual for the joystick and previously written "C" routines written by Captain Bob Filer that used the joystick (17). The user's manual detailed the capabilities of the joystick: what it would return, how it would return it, and how to control it. The objective in using these two sources of information was to find out what a joystick object was and what it would do.

The attributes of the Joystick class were specified first. The Microstick user's manual provided the information needed to specify the data that a Joystick object would have to encapsulate. Please refer to figure 4.2.

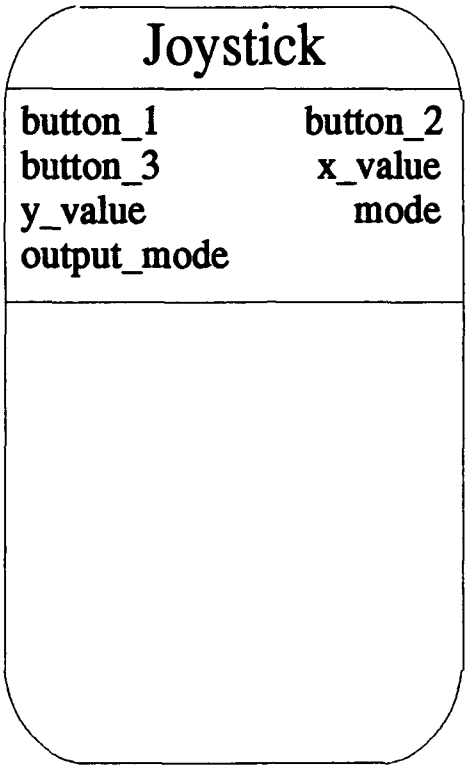


Figure 4.2. Initial Joystick Attributes

The attributes included: button 1 value, button 2 value, button 3 value, x value, y value, the joystick mode (it has 6 ways of presenting data), and the joystick output mode (polled, continuous or only when joystick values changed).

All six of Coad and Yourdon's tips listed in chapter 2 for identifying attributes were used including rule 2. Rule 2 asks "How am I described in this problem domain ?". This rule would appear to conflict with the stated design objective to construct classes for use by any application. The reason that it does not is that the objective of analysis is to produce a model of the "real world" problem. Adding methods for the sake of completeness (or some other reason) is best kept as an activity of design because these additional services do not exist in the problem at hand.

The methods of the Joystick class were done next. The methods were based upon the attributes that had been identified previously. The first methods that were

specified were the selector and modifier methods for the joystick mode and output mode. I decided not to supply separate selector methods for the joystick button and position attributes because they are returned at the same time by the joystick. Therefore, the characteristic "Read" function was given the job of returning all of them at the same time. It made no sense to add a modifier method for the buttons or the position values since the joystick offered no such capability. Please see figure 4.3.

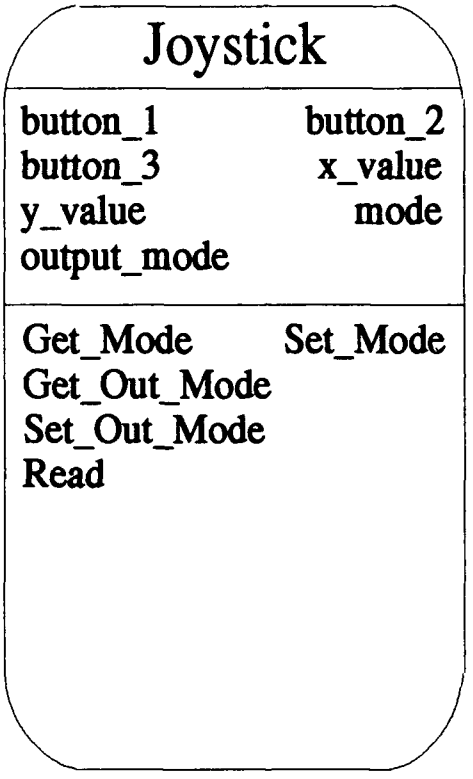


Figure 4.3. Initial Joystick Methods

4.3.2 High Level Design I The detailed analysis step produced a workable definition of what a Joystick object contained and what kind of capabilities it would present. The next step that I took was to specify how the data was going to get from the joystick and into the computer. This particular activity actually fit into two

parts of high level design: External Data Requirements and System and Structure Refinement (using existing code).

4.3.2.1 External Data Requirements and Reusing Existing Code The objective of the external data requirements step was to define how external data would be accessed by the computer. The Microstick user's manual specified that the joystick communicated with the computer through an RS232 port. In addition, Captain Filer's existing joystick routines showed how to utilize an RS232 port using UNIX system calls (17). Thus, the idea for the RS232 Port class was derived from these two sources.

I initially decided that the RS232 Port to Joystick relationship was a "generalization-specialization" relationship. This was based on the premise that the Joystick simply added more functions onto an RS232 port. Consequently, the Joystick was designated as a derived type inherited from the RS232 Port class.

Given that the RS232 port was accessed through UNIX system calls, work then shifted into how to reuse existing code within an object-oriented framework (part of the System and Structural Refinement activity). Captain Filer's code was again instrumental in that it detailed the UNIX system calls used to control an RS232 port (17). Using Captain Filer's code and three other sources on UNIX system calls (24, 44, 52), I was able to gain all the information necessary to construct a wrapper class for the RS232 port.

There were three activities to perform when constructing the wrapper: 1) identifying the system/procedure calls, 2) identifying the data that the calls used, and 3) adding necessary methods and encapsulation so that the resulting wrapper would be reusable. Using the existing code and references, I performed these three steps and identified the "Open", "Close", "Read", "Write" and "Flush" methods for the new RS232 Port class. I also identified the attributes that an RS232 port object would have to remember in order to be used correctly (chapter 6 discusses

the process of folding existing code into an object-oriented framework).

4.3.2.2 Concurrency There was a degree of concurrency indicated in the design. The joystick would, in certain modes, operate independently of the computer. In addition, the UNIX operating system was capable of multiple processes running at the same time (although in a timesharing mode). The UNIX operating system input routines associated with the RS232 port would be part of a separate process from the one running the simulator. Identifying concurrency in the high level design phase made the issue of concurrency easier to handle in the implementation phase because I knew where to expect it.

4.3.2.3 System and Structure Refinement Up to this point, one part of system and structural refinement had been completed — reusing existing code. Three activities remained: inheritance refinement, class refinement, and resolving multiple relationships. The question of inheritance refinement and resolving multiple relationships was not an issue at this point because there were only two very primitive classes in existence. The system and structural refinement centered on the class refinement.

The purpose of the class refinement phase was to examine each class with reusability in mind. The RS232 Port class did not appear to need any adjustments but the Joystick class needed some work. The Joystick class was not intended to serve as a base class so I looked at it solely in terms of what a user would want in a Joystick object. I examined the Joystick class with respect to the principles of sufficiency, completeness, primitiveness, clarity of design and simplicity (robustness is mainly a question of implementation and will be addressed in low level design).

The Joystick class seemed to provide sufficient methods with which to control the joystick. There were methods to access all attributes and two methods provided to change how the joystick sent information. The concept of "sufficiency" was used as a kind of lower bound on the amount of functionality to provide in an object.

The concept of sufficiency only requires that the designer has built in the minimum methods into a class.

Completeness was the other end of the spectrum. When I concentrated on completeness, I asked the question "What else would I want this object to do for me?". This line of questioning lead to the identification of the "Suspend" and "Resume" methods which, in effect, turn the joystick off and on. Because it was possible to get carried away with completeness, I kept the idea of "primitiveness" in mind.

The notion of primitiveness is offered by Booch as a check against getting carried away with completeness (3:125). The check for primitiveness entails examining each method and determining if a user can duplicate the method using simpler methods that the class already provides. The primitiveness check on the Joystick class revealed no such methods.

The principles of simplicity and clarity of design were not a problem either. The Joystick class had few attributes and a manageable number of methods. The names of the attributes were chosen to match what they actually stored. The methods were of standard "Get" and "Set" variety with a few other aptly named methods in addition to these. The Joystick class seemed clear and simple thus far.

4.3.3 Low Level Design/Implementation I The low level design activity was where the design finally took the form of code. Low level design was done with the strengths and weaknesses of C++ in mind. Using good design principles and designing for reusability were still of primary concern. The first iteration of low level design for the Joystick and RS232 Port classes involved all four phases of low level design: 1) deciding upon object representation, 2) implementing object methods, 3) establishing object visibility, and 4) identifying polymorphic methods.

4.3.3.1 Deciding Upon Object Representation and Implementation of Object Methods None of the attributes defined in the high level design of the Joy-

stick and RS232 Port classes needed to be implemented as an object. None of the attributes were important for anything other than the values they stored. There was no extra functionality associated with them. Accordingly, all of the attributes were implemented with standard C++ data types.

The high level design identified only one attribute for the RS232 Port class (the port number). The rest of the attributes defined in the RS232 Port class were motivated primarily by the data that UNIX needed for proper operation of the RS232 port itself.

One of the primary motivations for making a wrapper class is to make library or operating system routines easier to use. All the RS232 port needs is the file descriptor once it is opened. Most of the complexity occurs in the initialization of the port. Therefore most of my efforts centered on trying to hide the complexity of the RS232 port initialization.

Certain variables had to be initialized properly in order for an RS232 Port object to function. The C++ language offers the capability to initialize the attributes of an object when the object is created. This capability is provided through what is termed a "constructor" method (the converse capability is provided by the "destructor", which will allow the user to perform actions when an object is destroyed).

I was faced with two problems: 1) even though I could hide some complexity, some attributes would still have to be initialized correctly and 2) the lack of the capability in C++ to define a subrange as a data type would necessitate a lot of checking for improper values in the constructor before using values the user passed in. The solution to the problems of making the class easier to use and more robust was to use enumerated types.

Using the class was simplified by using descriptive names for each enumerated value that would be used to initialize the appropriate attributes (eg. "raw" and "canonical" for input handling). Using enumerated types also made the class more

robust because the compiler could now check for illegal values for the attributes.

One other attribute called "port open" was added to make processing a little smoother. It would make no sense to try to read a port that had not been opened yet. A method was also added to get the value of "port open". Adding this attribute made the class even more robust.

The last adjustments to the class were the use of default parameters. C++ allows the programmer to specify the default values for any parameter passed to any method. This capability was particularly valuable in implementing the constructor because about the only attribute the user would normally need to specify was the port number. All of the other parameters were not likely to be changed from a given default. The parameters were arranged such that a user would only have to specify the port number. All others would be set automatically.

The actual coding of the RS232 Port class methods were fairly straightforward. Four of the methods were one or two lines. The "Open_The_Port" method was considerably more complex than the others due to the way in which the port had to be set up and opened. I relied upon Captain Filer's code for guidance in this area (17).

The implementation of the Joystick class methods proceeded in a manner similar to the way in which the RS232 Port class was implemented. I made use of enumerated types to specify the joystick mode and the output mode. I also used default parameters in the constructor to cut down on the number of parameters that the user had to pass under normal circumstances. One difference between the implementation of the two classes was that speed was more of a concern with the Joystick. The "Read" method and the form that the attributes eventually took reflected this.

4.3.3.2 Establishing Object Visibility The first aspect of establishing object visibility, ensuring that communication can take place, was not a problem.

There were only two classes thus far and one was inherited from the other. Defining what was to be hidden inside each object was the main task.

The C++ programming language offers a number of keywords that a programmer can use to define what is hidden within an object. These keywords are all used within the class header. The four C++ keywords are "private", "protected", "public" and "friend" (55). An extended explanation of these constructs and suggestions for when these keywords should be used can be found in appendix E.

I started with the Joystick class. All data members were declared private (as opposed to protected) because I didn't expect the Joystick class to serve as a base class. The original implementation contained no extra methods that the object would use to implement higher level functions. Accordingly, all the Joystick methods were made public. With no private methods, there was no need for friend classes.

The next question was whether to make the inheritance relationship between the Joystick and RS232 Port classes public or private. Were there any methods of an RS232 Port that a user of a Joystick object would need to have access to? Because the Suspend_All_Inputs and Resume_All_Inputs methods were not declared as static, the inheritance had to be public.

Given that the inheritance was made public, attention turned to what methods to make visible only to the Joystick class. What would the Joystick need that someone using a Joystick object wouldn't? This included all but the "Suspend_All_Inputs" and "Resume_All_Inputs" methods, which were declared as public.

The last questions concerned the visibility of the data members of the RS232 Port class with respect to the Joystick class. After looking at the data members and the methods where they were used, I decided to make all of the data members of the RS232 Port class private. The Joystick class had no reason to access the attributes of the RS232 Port class outside of the methods offered by the Port.

4.3.3.3 Identifying Polymorphic Methods The only polymorphic methods that were required were the "Suspend" and "Resume" methods. This meant that the methods had to be specially designated in the base RS232 Port class as "virtual". Any method in a derived class that had the same name and parameters as the virtual method would be polymorphic. Thus, the "Suspend" and "Resume" methods in the Joystick class had the same name and parameters as their counterparts in the RS232 Port class.

The first implementation of the Joystick and RS232 Port classes was successful. All of the methods were tested and performed as intended. The object diagrams of the Joystick and RS232 Port classes as they appeared following their first implementation are depicted in figure 4.4.

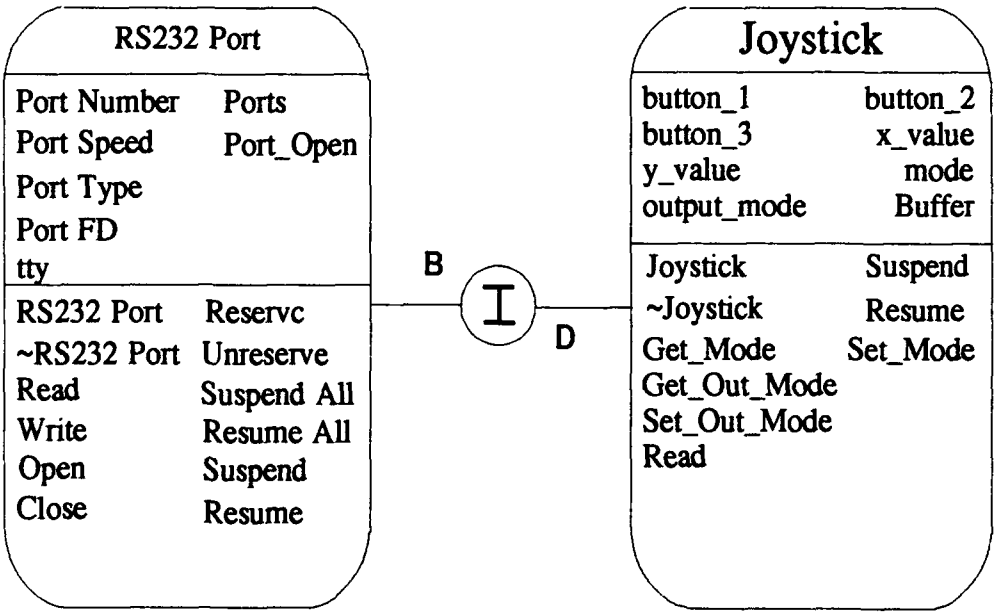


Figure 4.4. Initial Implementation of the Joystick and RS232 Port Classes

4.3.4 High Level Design II All of the activities of high level design were performed again in the second iteration. The activities previously highlighted in the explanation of the first iteration will be covered only briefly, if at all. The main activity of the second iteration of high level design was Inheritance Structure Refinement.

The second iteration of the design process was prompted by the requirement to make it possible to attach the joystick to a different computer than the one that was running the simulator. The most significant effect was that it highlighted the fact that the inheritance relationship between the Joystick class and the RS232 Port class was wrong.

4.3.4.1 Design Activities II The second iteration of the External Data Requirements activity was concerned with detailing how data was going to get from one machine to the other. Input on one machine had already been implemented with the two existing classes. The question was how data would get from one machine to the next. The answer was the UNIX socket.

Concurrency then became a bigger concern. Sockets enabled true concurrency in the system. The main goal was to identify the thread of control through the program. The solution was to specify two more classes: A "Port Reader" class and the "Distributed RS232 Port" class. Please refer to figure 4.5.

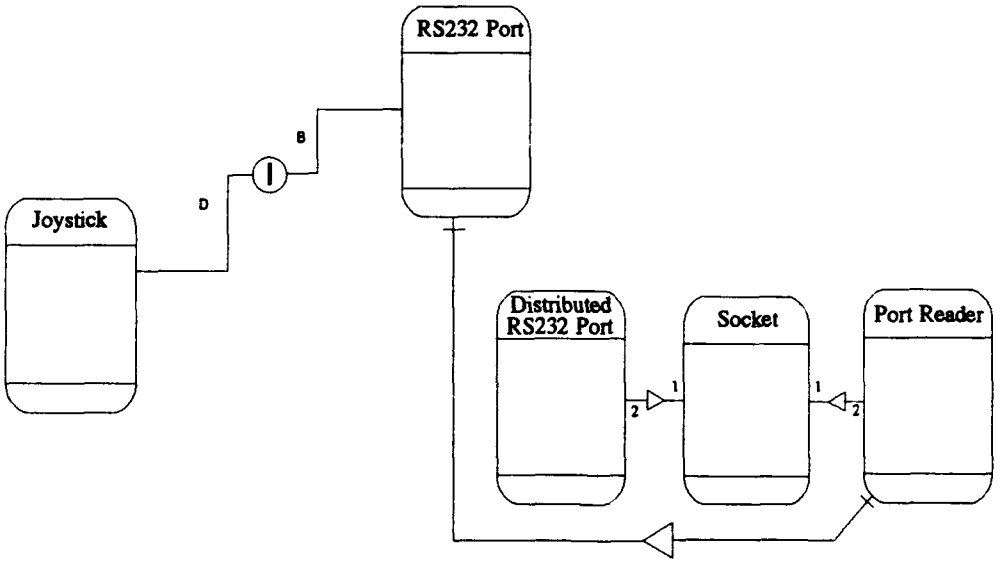


Figure 4.5. Second Design of Joystick and RS232 Port Classes

The Distributed RS232 Port class would be responsible for "kicking off" a Port Reader on the machine that would run the joystick. The Distributed RS232 Port was to act exactly as an RS232 Port class object. The fact that the information was coming from another machine was to be transparent. The Port Reader would do nothing but read the RS232 Port that the joystick would be connected to and respond to the Distributed RS232 Port. Communication between a Distributed RS232 Port object and a Port Reader object would be via Socket objects.

4.3.4.2 Inheritance Structure Refinement The main activity of the second iteration of high level design was Inheritance Structure Refinement. The purpose of the inheritance refinement phase in this example was to define how to best arrange the additional classes found in previous activities.

The main activities of this phase included finding commonality in the system, standardizing protocols, and identifying abstract classes.

The first task was to identify the commonality in the system. The main question was how to tie in the Distributed RS232 Port class and the previously existing RS232 Port class. The purpose of the Distributed RS232 Port was to provide a transparent interface with another machine. Therefore, it had to provide the same methods that the RS232 Port did. The way in which the methods could be implemented would be hidden within the Distributed RS232 Port.

This lead to the second step: standardizing protocols. The two Port classes had to offer the same methods. Thus, a root class simply named "Port" was created to standardize the methods that the two Port classes would offer to users. The Port class also contained the "port open" attribute that they both appeared to need. The RS232 Port and Distributed Port classes were designated as derived types of the Port class.

The Port class was designated as an abstract class. The Port class offered no functionality on its own. It would make no sense to have a Port object. It depends upon child classes to fill out the definition of the port.

The inheritance structure still needed a few adjustments at this point. The existing RS232 Port class methods used extra attributes to check the ports that were being used on a machine. I decided that the Port Reader would not benefit from using this extra checking, so I split the RS232 Port class into two classes.

The Unmanaged RS232 Port was the same as the RS232 Port class except that it lacked the extra port management. A derived class of the Unmanaged RS232 Port class called "Managed RS232 Port" would act just the same as the old RS232 Port class did. I also added a "Distport" object in recognition of the concurrency in the design. The Distport was an object (actually just a "main" program) that would make and run a Port Reader. The Port Manager object would be responsible for the

extra checking of port usage given different machines.

4.3.5 Low Level Design II — Revising the Joystick Class The Joystick was originally implemented as a derived class of the RS232 Port class. The new requirement to make the joystick readable from another machine prompted a major revision of the structure associated with the RS232 Port class. When the activities turned to implementation, the resulting structure highlighted the fact that the original inheritance relationship between the Joystick and RS232 Port classes was wrong. Revising the relationship encompassed all phases of low level design.

The Joystick needed to use both the Managed RS232 Port and the Distributed RS232 Port. Inheriting from the Port class would not work because it offered no functionality on it's own. It made no sense for the Joystick to multiply inherit from both the new Managed RS232 Port and Distributed RS232 Port because they offered the same methods. The only recourse was some kind of composition relationship.

The first option was to make each port a component of the Joystick. If I were using a language that did not support polymorphism, making both types of Ports components of the Joystick would have been a good option. However, matters were simplified by the polymorphism that C++ offered. I made a pointer to a Port class object a component of the Joystick class.

A pointer to a Port class object was now a component of the Joystick as opposed to the Port class acting as a base class for the Joystick. Given this relationship, the Joystick could instantiate any type of Port and use polymorphic methods to use any type of port. The only time the Joystick object would have to be aware of exactly which type of port that was in use would be when the Joystick object was created. It would be transparent as to what actual type of port was in use from then on.

Fixing the relationship between the Joystick and Port classes was the last major structural change to this portion of the design. The system diagram for the Joystick and related classes appears in figure 4.6.

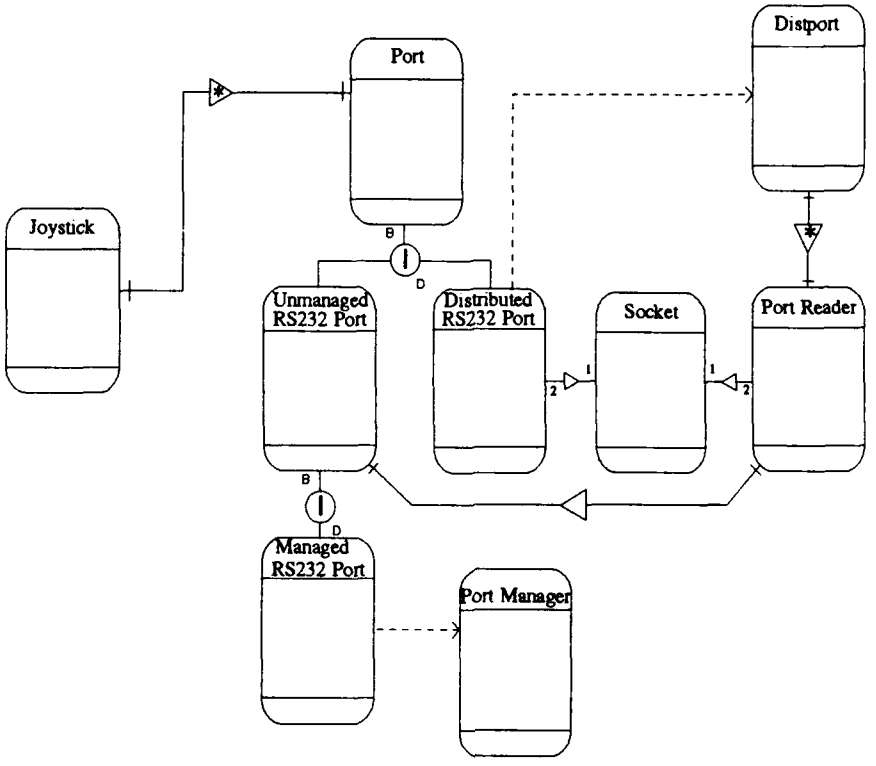


Figure 4.6. Full Design of the Joystick and RS232 Port Classes

4.3.5.1 Trouble Signs In looking back on the initial implementation, there were two signs that should have alerted me to the fact that I had picked the wrong relationship between the two classes. The first was the basic nature of the relationship. Was a Joystick really "a kind of" RS232 Port? Korson argues that, in a good design, root classes should be abstract forms of the child classes (28:54). An RS232 port is not an abstract form of a joystick.

I argued that the relationship between the two classes was a "generalization-specialization" relationship. I reasoned that the Joystick was just a more complicated form of an RS232 port. This was wrong. I had confused added functionality with true specialization. A true specialization relationship would have been one where I would have had to access all (or most) of the methods of the RS232 port outside of

solely using them inside the Joystick. This lead to the second indicator.

The second, and more definitive, indicator as to the true nature of the relationship showed up during the "Establish Object Visibility" phase of the low level design. I purposely hid all of the data members of the RS232 Port class from the Joystick class because the Joystick had no use for the data outside of methods that the RS232 Port class offered. In addition, the main methods of the RS232 Port were hidden from users of a Joystick object. These should have been the "dead giveaway".

A designer should never seek to hide data members of base classes from derived classes. This attests to the nature of the inheritance relationship. If a derived class is truly "a kind of" some base class, then the derived class should legitimately have access to all the data members of the parent. The portions of the base class that make up the derived class are an integral part of the derived class and not just a component.

Granting access to methods of base classes through derived classes should also be closely examined. In general, the characteristic methods of the base class should be public to users of any derived classes. Accordingly, trying to hide or "uninherit" methods is also a sign of an incorrect relationship.

The desire to hide data members or characteristic functions is a sign of an incorrect relationship. The amount of hiding that a designer wishes to attempt provides a general sense of how bad the relationship is. Trying to hide most or all data members and/or methods of a base class indicates that the relationship is more properly a component relationship. At a minimum, seeking to hide a significant amount of information from derived classes is an indication of poor factoring.

4.3.6 Final Design Activities The final design activities included adding functionality to existing classes. The need for these methods arose when the lower level classes were used by higher level classes. There were certain situations where the higher level object was performing actions that best belonged at a lower level.

Examples included controlling the noise inputs from the joystick, normalizing the joystick inputs and controlling the resolution of the inputs from the joystick. This was all added to the joystick to make it easier for the higher level objects to use.

4.4 Static Data Members and Static Methods

With C++, data members and methods declared as "static" can best be thought of as data and methods offered by the *class.* In this manner, the class functions as something more than a template for objects. The static data members of a class can be accessed and modified by all objects of the class. Similarly, all of the static methods can be used by the objects of the class without restriction.

Unfortunately, the current literature on object-oriented design and programming doesn't address how to consider static data members and methods with respect to the object-oriented paradigm.

The critical question in understanding their role was "What would I do if I didn't have static data members and methods?". The answer was that I would have to define another class to do the same job. Based on this observation I concluded that C++ static data members and methods should be considered as nothing more than another way in which to implement an object.

The "Port Manager" class/object indicated in figure 4.6 was based upon the realization that what was really encapsulated by the static methods of the Managed RS232 Port class was a "Port Manager" object. This discovery concerning the use of static constructs came in the middle of the effort. Therefore, I had to go back to existing classes and restrict access to all static data members. The only access to static data was controlled through static methods. This encapsulation made the static data and methods appear to be just like any other object in the system.

4.5 The Window and Text Window Classes

The Window and Text Window classes are an example of how to design classes that will be reused as base classes. The Window class was designed to encapsulate the information necessary to define and open a window. The Text Window class is derived from the window class and adds the capability to display text in a window. The main features of the two classes are that the methods are small, cohesive and self-contained (no coupling between methods).

The window class is a wrapper for the library routines used to define a window. It encapsulates the data and methods that are common to all windows of the system. It is an abstract class that contains a standard protocol for all windows that will be derived from it. It is an abstract class because it contains no information about what is to be displayed in the window. This will be provided by child classes that inherit the window class.

One facet of the methods that is not readily apparent is that they are all self-contained. The user need not worry about any particular sequence when using the methods other than using the methods in the right order to accomplish some higher level function. There is no hidden logic that applies across lower level methods to require that one be called before another.

The Window class is coupled to the "Window Manager" static object. The Window Manager maintains the identifier of the current window of the system. The current window is the one that the user has designated as such through the appropriate method "Make_Current" or through enabling automatic changes through the position of the cursor (a function of the Queued_Input Manager and the Window Manager). Simply using a method of a window that is not presently the current window will not change the window to being the current window.

All library calls apply only to the current window. Therefore, each method in the window class must change the current window to the window to which it belongs,

perform the required action and then change the current window back to what it was prior to the method call. This makes the methods self-contained. This also avoids confusion in that the only method that actually switches the current window after it has completed is the "Make_Current" method.

The Text Window class also contains methods that are self-contained. All of the methods that the Text Window class provides also do not require specific sequencing. For example, both the "Clear" and "Draw" methods correctly set the current window when they are executed. A user could call a Draw and then a Clear without having to worry about whether the current window was set properly.

The "Clear" and "Draw" methods were actually added in response to another programmer who was reusing the Text Window. The initial design included only the "Redraw" method. This proved to be too restrictive in that the screen did not have to be cleared before drawing the text in some situations. Therefore, the Redraw was split into the Clear and Draw methods to make using the class easier. This was such a good idea that the Draw and Clear methods were added to the standard protocol specified in the Window class.

4.5.1 Resolving Multiple Relationships The high level design activity of Resolving Multiple Relationships is the last phase to explain. This step in the high level design process focuses on how multiple relationships will be implemented in the system. Initial efforts directed at finding the structures present in the Window system are depicted in figure 4.7. A number of multiple relationships are indicated in the figure. The following discussions highlight how the multiple relationships were resolved within the Text Window and Window Manager classes.

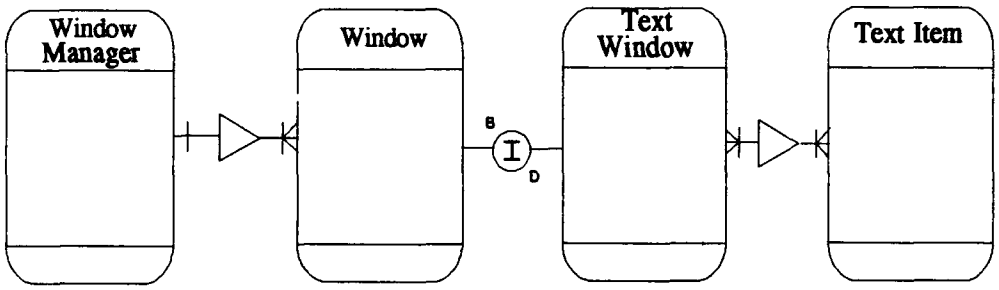


Figure 4.7. The Window Class Heirarchy

Each multiple relationship in the system was examined from the standpoint of the questions listed in chapter 3. Given how each list was going to be used, I made decisions on how best to represent the list in the design. Specifying these lists sometimes involved the addition of attributes to classes and in some cases, the addition of new classes to handle the list. With regard to the relationship between a container and the contained objects, deciding whether the multiple relationship is a composition relationship or interobject communication is based upon the need for the containing object to use the methods of the objects that are being contained. If the container needs methods from the objects being contained, then the container must be composed of the objects being contained. Otherwise, interobject communication will suffice.

The relationship between the objects being contained and the container is

either non-existent or one where the contained objects communicate with the container. The ideal situation is where there is no communication because this means there is no coupling between the objects of the list and the list itself. This is often the case where a third class of object is responsible for managing the list.

The second situation occurs where the objects being contained add and/or delete themselves from the container. While this denotes extra coupling in the system, it may be justified by added functionality made possible by the arrangement. This was the case with the relationship between Window objects and the Window Manager.

The Window Manager exists to provide extra functionality to the windows. Through it's communication with the Queued Input class (not pictured in figure 4.7), the Window Manager provides automatic redrawing of windows and changing of the input focus. The Window Manager also tracked what the current window was. The Window Manager is able to provide this functionality by keeping a list of all of the windows that are open at the time.

The Window Manager was required to call methods of the windows that it contained, so this was a composition relationship. The Window class did not depend upon any other class to add or delete windows from the Window Manager, so there had to be a way for a Window object to send messages to the Window Manager.

Given that the relationships were correct, deciding what type of list to use came next. The Window Manager list was going to be accessed often. It was also important that access to particular windows be granted quickly. There was the possibility that windows could be opened and closed quite frequently as well. However, there was no significance as to the relationship between windows themselves (like there would be in a generalized list). Given this, and the fact that each window already had a unique identifier, I decided to specify that the windows would be stored in a hash table in the Window Manager class.

The relationship between the Text Window and Text Item classes was very different than the relationship between the Window class and the Window Manager. A Text Window was composed of many Text Items. The Text Window used methods of the Text Items in order to display them. In contrast, a Text Item object did not have to be aware that it was part of a Text Window. This is because the Text Window or some other class of object was responsible for placing the Text Item in the Text Window.

There were two types of Text Items: those that would change while the Text Window was opened (dynamic text) and those that would not change (static text). This slightly complicated matters in that this necessitated two lists within one text window. The two different lists were required because of the differences in the way the static and dynamic Text Items in the Text Window were going to be processed.

Since each Text_Item contained a unique identifier, I again chose a hash table within the Text Window to store the dynamic Text Items. Because the Static Text items had less stringent requirements upon their usage, I decided to store them in a singly linked list. The decision on which lists to use also drove the addition of an attribute to the Text Item class.

Each type of list needed some kind of link from one Text_Item to the next. This had to be associated with each object in the list. Thus, I added a "next_item" attribute to the Text_Item class and services to "Get" and "Set" the attribute. This enabled the object to serve as part of either type of list of Text_Items.

The purpose of this step was to specify how multiple relationships were to be represented in the design. Choosing a particular type of list was based upon how the list was going to be used. The intent of this phase is to specify these relationships at a high level of abstraction so that the list could be implemented in any language.

4.6 Conclusion

The purpose of this chapter was to exemplify the methodology presented in chapter 3. The principles in chapter 3 are not meant to be applied in a rigid sequence. Despite the fact that some steps in the process must initially precede others, iteration through all phases of the design is a feature of the object-oriented paradigm.

The methodology was presented by using the development of the Joystick class as an example. Designing the class was done with reusability and the design strategy to build for speed in mind. These two principles permeated the process and drove many of the design decisions. The example was arranged to show where each activity performed in the implementation of the Joystick fit into the methodology. The most important aspect of the example was that it showed how and why the Joystick was taken through three iterations of the design process along the way to its final completion.

Two aspects of the methodology that were not exemplified in the explanation of the Joystick class were designing a class to be reused as a base class and resolving multiple relationships in the high level design. The Text Window and Window classes were used to represent these two concepts and finish the coverage of the ideas contained in chapter 3.

V. Reusing Procedurally Oriented Code

5.1 Introduction

This chapter will discuss the techniques that were used to fold procedurally oriented code into an object-oriented framework. The technique will be illustrated by using various wrapper classes as an example of how the technique was applied. The technique will work equally well for code that is taken from normal "C" programs as well as system/library routines like the ones used in the wrapper classes.

Folding procedurally oriented code into an object-oriented framework encompasses both high level and low level design. The high level design aspects include identifying the entity to be encapsulated and the methods that it will offer. Implementing the methods and attributes is a part of low level design. Admittedly, the line between high level and low level design in this process is very thin.

The first section of this chapter will discuss the motivation behind constructing classes from procedurally oriented code and will set the stage for the explanation of the technique itself. The technique consists of three steps: 1) identify the candidate class, 2) identify the data and all relevant procedures that use the data, 3) implement a class to encapsulate the data and procedures. Each step will be explained in a separate section of this chapter.

5.2 Reasons for Folding Procedures into Classes

There are two basic types of sources of procedurally oriented code: 1) other programs, and 2) system/library routines. Using data and procedures from other programs is more flexible because the implementation of the actual procedures and data can be changed. The designer can copy (or translate) the existing code and then basically do anything with it. This differs from using system/library procedures.

The implementation of the system/library routines and the data they operate upon cannot be changed, so they must be used "as is".

The main motivation behind using existing code is reuse. This includes reusing existing code to save time with the present programming effort and making the code more reusable in the future by encapsulating it within a class. Folding procedurally oriented code into an object-oriented framework is done to hide complexity, offer more functions, and isolate system/library calls to one location.

One of the main principles of the object-oriented paradigm is encapsulation. By hiding the complexity of using procedurally oriented code within a class, the class becomes easier to understand and reuse. The user only has to concentrate on using the new methods of the class and not on how they are implemented. This principle applied to both sources of procedurally oriented code but was particularly applicable to putting wrappers around UNIX routines.

Extending the functionality of existing routines was also a key factor in making the new classes more reusable. For example, UNIX routines are not normally useful in isolation. They are usually put together in a specific manner to provide a useful function. Part of the process of putting a wrapper around UNIX routines was to identify the higher level functions that would be offered as methods of the new class. Implementing the methods was accomplished by using the smaller UNIX functions.

The last reason for making a new class around procedurally oriented code concerns localizing system/library calls. If system/library routines calls are localized within one class, then switching to a different operating system and/or machine would entail changing only that one class. A designer would not have to hunt through the rest of the code in order to find what had to be changed in order to switch machines.

5.3 Identify the Candidate Class

This was the starting point of the technique. It was performed during the high level design phase of the project. The purpose of this step was to identify a candidate class. Identifying what to encapsulate focused the efforts in future stages.

The need to encapsulate existing code was driven by the needs of the application and what was available to reuse. For example, the Socket class was originally identified because there was a need for the Flight Simulator to communicate with processes running on different machines. UNIX sockets were the only available option. Other classes of the Flight Simulator that were identified in a similar manner included the "Window", "Queued Input", and the original "RS232 Port" classes.

The motivation for the identification of the candidate classes were the requirements of the application. There are many other classes that could have been based upon UNIX routines but they were not needed for the Flight Simulator. The point is that a designer should be aware of what the needs of the design are vs. everything that could possibly be done with existing code.

A designer should not become overly concerned with the mechanics of the procedures themselves at this stage of the design process. Examining the specifics of the existing routines is an activity of implementation. The focus at this stage should be on the general capability offered by the procedures and the possible methods that might be derived from them.

During this stage, the designer should focus on the general aspects of the problem so that the resulting class doesn't turn out to be nothing more than a collection of existing routines. The end result of this process is an object that encapsulates as much detail as possible. The procedurally oriented code that is to be encapsulated may not be based upon information hiding principles. Focusing on the details of the available procedures too early will lead to an object that doesn't hide as much information as it could.

Consider the Socket class. I had originally identified it by researching UNIX interprocess communication facilities. Sockets were the only way to communicate with processes on other machines. Once I had identified the entity that I wanted to encapsulate I tried to define the methods that a Socket would offer in the Flight Simulator. I reasoned that the Socket class would offer six methods given this application: "Read", "Write", "Open", "Close", "Listen" and "Connect". The only attribute that I defined at this point was the "socket number".

It is likely that there will not be a multitude of attributes defined in this stage. In fact, there may not be any attributes that are apparent at this point. The main reason for this is that the data encapsulated by the class is highly dependent upon the procedures that use the data. However, some very general attributes should be apparent. Including these general attributes is advisable in order to keep in mind that the end result of the process is an object and not merely a collection of procedures.

The end result of this step is a candidate class that will be implemented (mostly) by using existing code. The objective is to define an object-oriented framework that is based upon the general capabilities of the existing routines and the requirements of the application. It is very important that the existing procedures be viewed only in very general terms so as not to compromise the principle of encapsulation. The candidate class represents an object even though it may apparently lack sufficient attributes at this stage. The idea is to define a candidate object and not simply a collection of related procedures.

5.4 Identifying the Data and Procedures

Once the high level design of the candidate class had been prepared, the process moved to implementation. The next step in the process was to identify the specific data that could be encapsulated in the class and all relevant procedures that could use the data. Identifying data and procedures was done in the same activity because

they normally can't be done in isolation from one another. The objective was to generate a group of related data and procedures that could be used to implement the desired methods of the candidate class.

Some of the procedures will already have been identified in the previous step. Analyzing their function in terms of the capabilities that they provided was the basis for the methods of the candidate class. These initial procedures form the starting point for a more exhaustive search directed at finding all possible data and related procedures that could be used in order to implement the candidate class.

The easiest way to approach the task was to begin with one procedure that was known to operate on the desired construct. Given this starting point, I found the data that the procedure used. If the source of the procedures is existing code, the next step is to find out where else in the existing program the data structure(s) used in the initial procedure are used. Finding other procedures that used the data might reveal other data. The process should be continued until no new data or procedures are found.

Identifying data and procedures must be done a bit differently if the source of procedures is system/library routines. The source code of these routines is not normally available to the designer. The best source of information on system/library routines are reference manuals. Fortunately, UNIX has an automated manual system that served this purpose quite nicely. The manual pages detailed the data that each procedure worked on and any other routines that were related to the one I was currently investigating. The end result was the same: a complete list of data and procedures that used the data.

5.5 Implementing the Methods

The last step of folding existing code into an object-oriented framework was to implement the candidate class. The high level description of the candidate class and the list of related data and procedures were used to finally put the class into code.

This involved implementing the high level class description, resolving unavoidable coupling, and making the class more reusable.

The first activity was to implement the methods that were outlined in the high level design phase. The list of procedures from the previous step made up a "shopping list" of building blocks to use in constructing the methods of the class. As each procedure was utilized, the data that it used was considered for inclusion as attributes of the class.

Deciding if data used by a routine should be included in the attributes of the class was based upon whether the data was going to be used again in some future action. For example, the file descriptor returned from successfully opening an RS232 port was used in all of the other system routines used in that class. Thus, it was saved as an attribute of the class. In contrast, the buffer used to return data from a read function was not saved because no other routines needed the data that was stored there. The designer must discriminate between parameters used only in the function in question and parameters that are shared with other related functions. Shared data must be included in the attributes of the class.

The initial implementation may contain code that is more properly included in other classes of the design. This is not a major problem when using code from an existing program. Unfortunately, the situation is sometimes unavoidable when using system/library routines. Some routines may be coupled to other routines that don't really fit into the class. If existing code is being used, it can be modified as necessary. If system routines are being used, then this unavoidable coupling must be dealt with.

An example of this situation occurred between the Queued Input and Window classes. The problem was that all of the library routines that were contained within the Queued Input class could not be used unless a Window was opened. The initial attempts at the Queued Input class used a routine that sampled what the current window was. This routine was included directly in the class in these first attempts

because the Window class had not been made yet. Once the Window class was made, the call to the window function in Queued Input was taken out and replaced with a method provided by the Window class. This put the function that sampled the current window into the class where it belonged and also localized its usage to one class.

The objective of managing unavoidable coupling is to hide all of it from the user of the class(es). The Queued Input Manager and the Window Manager objects were eventually used to perform functions to manage the objects of their respective classes. They are tightly coupled objects. This coupling is a consequence of the tight coupling of the library functions used to implement the Queued Input and Window classes. All of the communication necessary to implement the two classes is done through the Manager objects and is transparent to users of either class. The point is that the coupling has been managed and hidden.

Once the class had been initially implemented and the inherent coupling effectively managed, the last step was to make the class more reusable. The class definition produced in the high level design phase may or may not have included all methods that would be useful for the class. The design at that phase was concerned with the general capabilities of the existing functions. The designer should know all the "building blocks" to use with the class from the second step of this process. Additional methods may become apparent given the full list of functions to use on the class.

After the additional methods have been done, every attempt should be made to hide even more complexity from the user. For example, I made use of enumerated types to hide the complexity of the RS232 Port class attributes and to make the class more robust. Allowing the user to specify "raw" or "canonical" mode versus requiring a value of 0 or 1 is an example. Another example was where I defined an enumerated type to specify if a socket was to be "read only", "write only" or "both".

5.6 Conclusion

The main motivation for using existing code in the design is reuse. This concerns reusing the code in the present design and making it more reusable for future efforts. The first step in the process was to identify the class that was to be implemented. This was done in the context of the general capabilities of the existing functions and the needs of the application. The result was an object-oriented framework within which existing routines would be used. The next step was to find all relevant existing functions and the data that the functions used. The class was then implemented using this "shopping list" of functions. The finishing touches of the process were to add additional methods and make the class easier to use.

VI. Summary and Recommendations

This chapter summarizes the research presented in this thesis and also makes recommendations for future studies.

6.1 Summary

The objective of this thesis was to present a comprehensive object-oriented design methodology. The methodology was applied to the implementation of a low cost flight simulator using the C++ programming language.

6.1.1 Literature Review The literature review in chapter II contained a thorough overview of the object-oriented paradigm. The object-oriented paradigm is not yet mature. Thus, there are many different views as to what the paradigm actually is and what it's elements are. The information in chapter two is an attempt to arrange these sometimes disparate views within a common framework.

The software crisis is the motivation for using object-oriented techniques. The object-oriented paradigm can be used to help manage the complexity associated with today's software projects. The elements of the object-oriented paradigm are the object, the class, the relationship between classes and interobject communication.

The building blocks of object-oriented systems are "objects". An object is a computer representation of a like object from the real world of the problem domain. An object encapsulates both the data and the methods that act upon that data. The data hidden within an object can only be changed or accessed through the methods provided by the object. An object is a dynamic entity that exists only when the computer program is running.

A class is a decription of the data and methods that make up the objects of the class. Whereas an object is a dynamic, runtime entity, the class is a static description

of the characteristics that all objects of the class will share. A class is a template for objects.

Classes can be related to each other by inheritance or by composition. If a class inherits from another class, then the child class will incorporate all of the attributes and methods of the parent class and will add it's own unique properties (47:3). An inheritance relationship can best be characterized by saying that the child class "is a kind of" the parent.

The second type of relationship between classes is the composition relationship. In this relationship, an object of one class is an attribute of another. One class "uses" another. This kind of relationship can be thought of as a "whole-part" relationship (10:91).

The fourth element of the object-oriented paradigm is interobject communication. Commonly referred to as "message passing", interobject communication is the way in which an object-oriented system functions. A "message" consists of one object requesting a service from another.

The object-oriented paradigm is based upon the principles of abstraction and encapsulation. The principle of abstraction is used to focus upon the pertinent details of a problem while ignoring those that are not important at the time. Abstraction allows the designer to focus on the outside of an object while ignoring what goes on inside.

The principle of encapsulation is concerned with hiding what goes on inside an object. Information hiding is a central feature of the object-oriented paradigm. The object-oriented paradigm supports information hiding through the idea of a defined interface within which to affect the state of an object. This interface is the methods offered by the object. The methods are the only way to affect an object and the implementation of the methods is hidden from the user.

The object-oriented software development lifecycle consists of analysis, design

and implementation phases. The object-oriented paradigm is seamless in that the products of each phase are used directly in the next. No translation of results is necessary from one phase to the next. While each phase builds upon the previous one, object-oriented software development is meant to be an iterative process.

The objective of object-oriented analysis is to produce a model of the real world in object-oriented terms. There are varied methods to approach OOA but they all shared some common themes. The OOA process consists of identifying the classes/objects in the system, identifying the relationship between classes, identifying the attributes and methods of each class and specifying interobject communication. Each of these steps can be applied in any order (and many times) along the way to producing a model of the system.

The results of OOA feed directly into object-oriented design. The objective of OOD is to decide how to move the "what" of analysis into the "how" of design. The results of OOA are added to and in some cases modified in order to define how the problem will be done by a computer. An example of this would be adding a class to handle some external source of data. Two methods of OOD authored by Coad and Yourdon (11) and Rumbaugh, et. al. (47) were summarized in chapter two.

Various authors have defined the elements of a good design. The majority of these measures and guidelines revolve around the principles of coupling and cohesion. Coupling is a measure of the interconnectedness between units in a system. The less coupling the better. Cohesiveness relates to the function that a unit performs in a system. Cohesive units perform one, focused function. A good design has a high degree of cohesion in each unit.

Object-oriented programming is concerned with the implementation of the system. Simply stated, OOP is programming with objects. The results of the design are used as the blueprints for the coding of the system. An object-oriented programming language should be used to implement the design.

The benefits of the object-oriented paradigm are that it provides a way to manage the complexity inherent in today's software, provides a "seamless" software development methodology, promotes reusability and promotes maintainability and extensibility. All of these benefits are derived from the principles of abstraction and encapsulation that the object-oriented paradigm is based upon. The classes, objects, relationships between objects and the way the objects communicate are the embodiment of these two principles.

Three drawbacks to object-oriented software development are performance considerations, startup costs and lack of direct support for constraints present in the system. The nature of object-oriented systems is such that they may not be suitable for applications that demand speed. Numerous methods calls, dynamic memory allocation and late binding all contribute to this problem.

The object-oriented paradigm is very different from traditional methods. Transferring to object-oriented practices from a more traditional procedurally oriented methodology means a fundamental shift in the way that software is produced in an organization. A large investment in training is necessary before jumping headlong into object-oriented development.

The object-oriented software development methodology does not provide any direct support for the constraints present in a system. Constraints are relationships between objects that must be enforced when the system is operating. Presently, the only recourse is to program these dependencies directly into the methods of the objects in the system.

Despite being in it's relative childhood, the object-oriented approach has been used to field systems in many different problem domains. Object-oriented techniques have been used in making compilers, air defense simulations, control programs, debuggers, CAD systems and user interfaces. Object-oriented techniques are in wide use today and it is expected that they will gain even more popularity in years to come.

6.1.2 Methodology Chapter three presented the methodology used to implement the flight simulator. The methodology is comprehensive in that it describes the phases of the object-oriented software development process from the initial problem definition through to implementation. There is no restriction that a phase can only be done once nor that it must be done completely. The object-oriented software development process is meant to be iterative and incremental (3:189). An outline of the methodology contained in chapter three is presented below:

I. Analysis

- A. Context Diagram
- B. Identify Classes/Objects
- C. Identify Structures
- D. Identify Object Attributes
- E. Define Methods

II. High Level Design

- A. System and Structure Refinement
  1. Class Refinement
  2. Identifying Code to Reuse
  3. Resolving Multiple Relationships
  4. Inheritance Structure Refinement

B. Concurrency

C. External Data Requirements

III. Low Level Design/Implementation

A. Object Representation

B. Implement Object Methods

C. Establish Object Visibility

D. Identify Polymorphic Methods

The process begins with analysis. The first task in analysis is to perform a context analysis of the problem at hand. Among other things, the context analysis defines the boundaries of the problem. The other four parts of the analysis are derived from Coad and Yourdon's method of OOA: 1) Identify classes/objects, 2) Identify structures, 3) Identify object attributes, and 4) Define object methods (11). The resulting analysis is a model of the real world and serves as the basis for the design.

The overall design process is dominated by practices that are used to make the classes of the design easier to reuse. These include the guidelines for a good design contained in chapter 2. Use of these guidelines helps to ensure low coupling, high cohesion and a high degree of encapsulation in the design. Making a class easy to reuse refers to making instances of a class easier to reuse and making potential base classes easier to reuse.

Making instances of a class easier to reuse includes hiding as much comlexity within the class as possible. The methods of the class should be complete, sufficient, primitive, clear and simple. The methods should provide all meaningful services given only the information contained within the object.

Making base classes easier to reuse is based mainly upon making the methods of the base class (and all its ancestors) cohesive. Making the methods cohesive will

help to avoid the requirement to duplicate code between classes in order to perform similar functions. The objective is to design base classes such that there will be many cohesive "building blocks" that a future designer can choose from in order to make higher level methods.

The design process is divided into high level design and low level design/implementation. The products of the analysis represent the real world. The objective of the high level design phase is to bring this model into a form that is suitable for implementation on a computer. The high level design phase consists of three major steps: 1) refinement of the system and its structure, 2) recognizing concurrency, and 3) accomodating external data requirements.

Refinement of the system and its structure consisted of four main activities: 1) class refinement, 2) identification of existing software, 3) resolving multiple relationships in the system, and 4) refinement of inheritance hierarchies. Class refinement involves looking at each class in the analysis with respect to making it more reusable. Identification of existing software is done to reduce the overall effort and to begin the process of folding existing routines into an object-oriented framework. Resolving multiple relationships is done to determine how multiple relationships will be represented in the system. Refinement of inheritance hierarchies is done to standardize protocols within inheritance trees and to organize the design in order to take advantage of similarity between classes.

The second phase of high level design revolves around identifying the possible concurrency in the system. This was essential in determining the type of control present in the system. This step is done with the capabilities of the operating system, implementation language and available hardware in mind. The decision to include concurrency should be driven by the needs of the application.

The third phase of high level design is designed to accomodate the external sources/sinks of data that the system must deal with. This includes external devices such as joysticks and mice in addition to data from files on the computer system.

The analysis will not generally include classes to bridge the gap between the external data sources and the computer. Specifying these "bridge" classes is the purpose for this step in high level design.

High level design is followed by low level design/implementation. The purpose of low level design is to implement the design given the capabilities of the implementation language. The four activities of low level design are: 1) deciding upon object representation, 2) implementing object methods, 3) establishing object visibility and 4) identifying polymorphic methods.

Deciding upon object representation is done in light of what methods that the candidate object must provide. If the object doesn't provide any methods beyond retrieving or saving components, then it may be acceptable to implement the object as a data type. If any further methods are needed, then implementing the candidate object with a class is required.

Implementing object methods is done with reusability in mind. Attributes can be added as necessary to improve the execution of the methods. Additional methods that are used only by the class to implement the object can also be added. Using enumerated types to make using the class easier to use was particularly effective when implementing the flight simulator.

Establishing object visibility consists of making sure that required data paths exist and specifically hiding the elements of an object. Establishing object visibility means making sure that the necessary communication can take place within the system given the capabilities of the language. Hiding information within an object deals with specifying exactly what a user of the object can see/use. This is also language dependent.

Identifying polymorphic methods is included in low level design because polymorphism in an artifact of implementation. Polymorphism can be viewed as another way to implement a method. Identifying polymorphic methods is done to set them

apart from "normal" methods in the system (some languages may require that polymorphic methods be specially designated).

6.1.3 Applying the Methodology Chapter 4 was devoted to providing examples of how the methodology described in chapter 3 was applied to the flight simulator. Each phase of the methodology was exemplified in chapter 4. The majority of the chapter centered upon the development of the Joystick and related classes. The explanation highlighted the fact that object-oriented development process is both incremental and iterative.

6.1.4 Reusing Procedurally Oriented Code The main motivation behind using existing code is to decrease the effort required in the new project. Existing procedurally oriented code and system/library procedures are folded into object-oriented frameworks in order to hide complexity, offer more functions and to isolate system/library calls into one location. The objective is make the reused code easier to (re)use and understand.

Reusing existing code encompasses both the high level and low level design phases. The candidate class is identified during the high level design phase. The designer should research the available code in order to get a general idea of the functions and the data that the functions work on. The result of the high level design should be a candidate class that will be implemented mainly by reusing existing code.

The objective in the high level design phase is to concentrate upon the general capabilities of the existing code so that the resulting object doesn't turn out to be nothing more than a collection of related procedures. The procedures should be viewed as potential building blocks for a class that will hide the complexity of using the existing code and data. Taking this conceptual step away from the procedures themselves and viewing them in a general sense should help to ensure that the maximum amount of information is hidden inside the class.

The low level design phase involves implementing the methods of the candidate class by using the existing code. Attributes are added to the class as needed. Under no circumstances should the user of the object have to track information shared between the methods of the class. Such information must be hidden inside the object. Final implementation also involves making the class easier to use by hiding as much information as possible from the user. Allowing the user to use enumerated types versus the exact data that a system call needs is an example of making the class easier to reuse.

6.2 Conclusions

One of the objectives of this thesis was to determine whether object-oriented techniques could be successfully used to implement a flight simulator. We wanted to end up with a system that was reusable, maintainable and extensible. Object-oriented techniques promised to provide a way to achieve these goals. I believe that these goals have been accomplished.

The object-oriented paradigm provided the tools with which to create a system that would satisfy the three requirements. However, it was still possible to create a system that was not what we required. It is just as easy to create a terrible system using objects as it is with procedures. It took the careful application of a coherent methodology to ensure that a well designed system emerged at the end of the process.

6.3 Recommendations for Future Research

There are many different areas of the design that could be extended and improved. The design has fulfilled its purpose in providing a baseline for future research. Some of the aspects of the flight simulator that could be extended are the interaction between simulators over a network, an improved aircraft cockpit display, and the addition of different types of entities in the system.

This research has addressed only the process of object-oriented software construction. Further research into the object-oriented software development methodology should focus on the products of each phase of the process described in this thesis. While I did introduce an extended notation to graphically depict the results of each step, a notation is by no means a complete (or sufficient) record of the process.

Very few sources that I used in this research detailed the products and documentation of the process (3, 10, 11, 47). Some work has already been done here at AFIT in formalizing the products of the analysis process (6). It would be beneficial to extend this type of approach into the areas of design and implementation. Work could also include such areas as class libraries and coding standards.

6.4 Remarks

Some authors argue that object-oriented design is a creative process and thus is not suitable for "cookbook" approaches (3, 28). I believe the term "creative" is used in reference to computer science when no widely applicable method is available to use. Thus, in the absence of some sort of method, the process becomes "creative".

I have not tried to produce a "cookbook" for object-oriented design. What I have tried to do was present process "stepping stones" that could be used to ensure that a system was well constructed from the problem definition all the way through to implementation. The boundaries between the phases of the object-oriented software lifecycle are not distinct. Hopefully the comprehensive view of the overall process of object-oriented software construction that I have presented should help to better define the different phases of the process.

Appendix A. *Bibliography of OOA Sources*

Bailin, Sidney C. "An Object-Oriented Requirements Specification Method," *Communications of the ACM, 32:* 608-622 (May 1989).

Booch, Grady. *Object-Oriented Design With Applications*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.

Booch, Grady. "Object-Oriented Development," *IEEE Transactions on Software Engineering 12:* 211-221 (February 1986).

Coad, Peter and Edward Yourdon. *Object-Oriented Analysis, 2nd ed.* Englewood Cliffs, NJ: Yourdon Press, 1991.

Henderson-Sellers, Brian and Julian M. Edwards. "Object-Oriented Systems Lifecycle," *Communications of the ACM, 33:* 142-159 (September 1990).

Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

Shlaer, Sally and Stephen J. Mellor. "An Object-Oriented Approach to Domain Analysis," *ACM SIGSOFT Software Engineering Notes, 14:* 66-77 (July 1989).

Appendix B. *Flight Simulator Context Analysis*

Problem Statement: The Flight Simulator should simulate flying an aircraft. The system should allow a user to realistically control the simulated aircraft. The system should provide three dimensional "out the cockpit" views of the imaginary world in which the user is flying. The system should have the capability to interact with other simulators running on separate machines.

Concept Map:

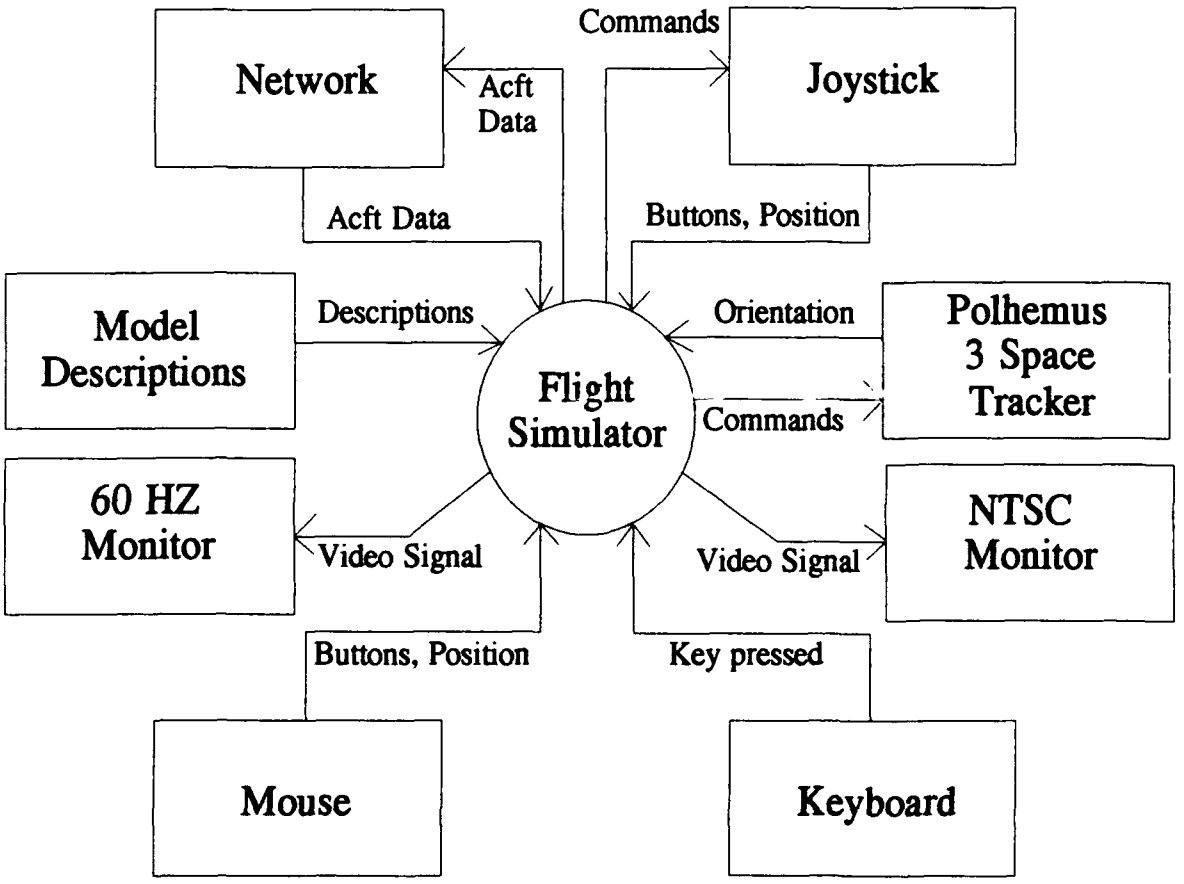


Figure B.1. Flight Simulator Concept Map

Event List:

— Aircraft data is received from the network

— Aircraft data is sent over the network

— User asks for help

— User changes the flap setting

— User changes the position of his/her head

— User changes the spoiler setting

— User changes the throttle setting

— User fires a weapon

— User moves the aircraft control stick

— User pauses the simulator

— User quits the simulation

— User resets the simulation

— User selects a weapon

— User toggles the landing gear

— User toggles the video output

Narrative Constraint List:

Economic:

— Existing equipment must be used to implement the system. Limited funds will be available for low cost (less than $1000.00) items if necessary.

Technical:

— System must be implemented using a Silicon Graphics 4D85 GT/GTX workstation.

— System must use available equipment for implementation — Polhemus 3 Space Tracker, CH Products Microsticks, AFIT Head Mounted Display and possibly the Dimension 6 Spaceball.

Legal:

— If code is reused from existing SGI programs, written permission must be obtained from Silicon Graphics to distribute the code to users outside AFIT.

Appendix C. *Flight Simulator Design*

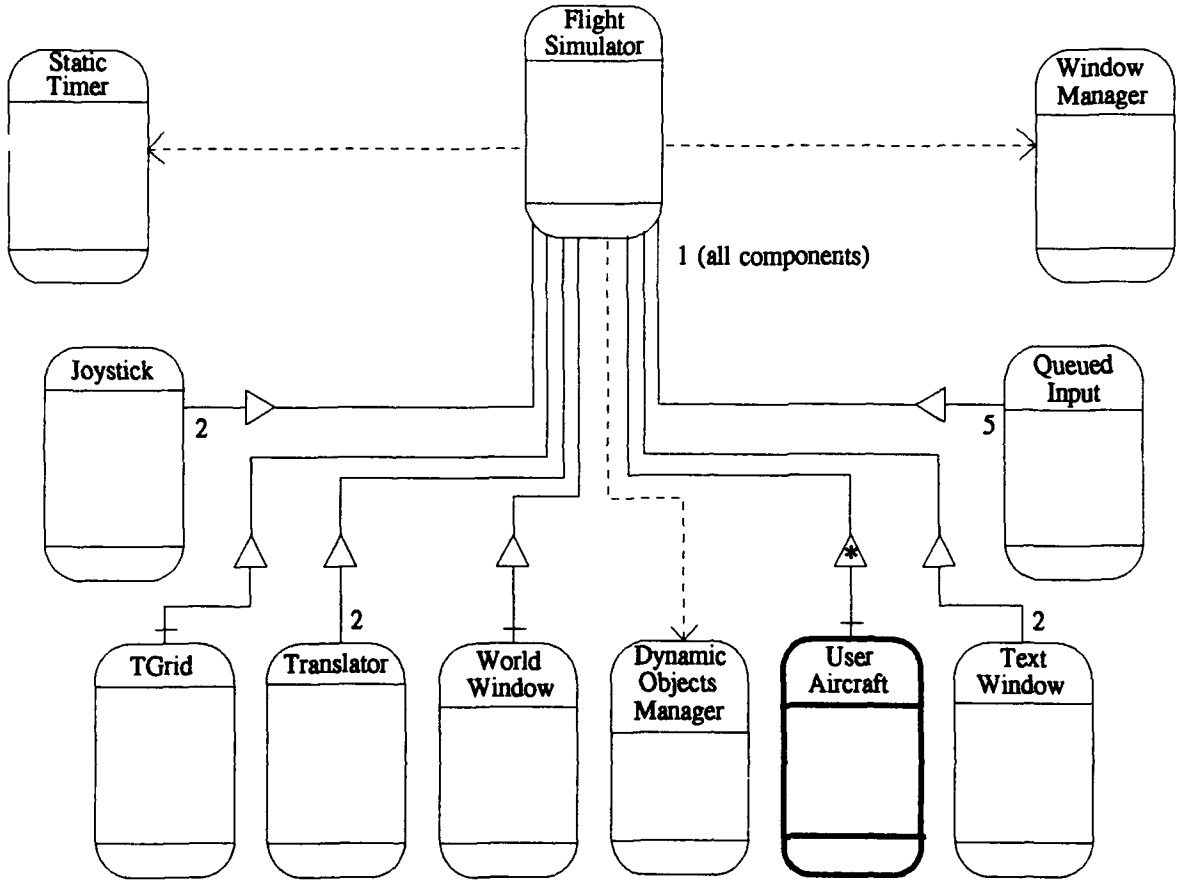


Figure C.1. Flight Simulator Composition

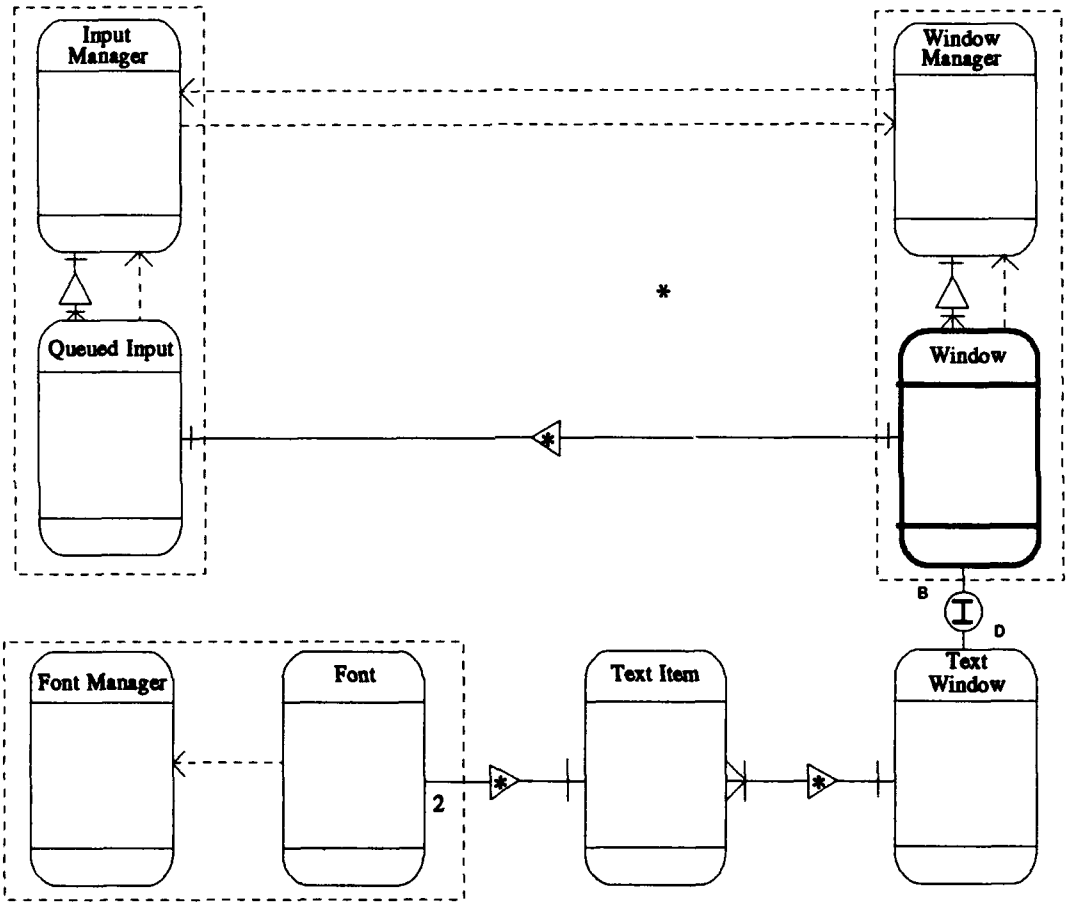


Figure C.2. Window and Queued Input Classes

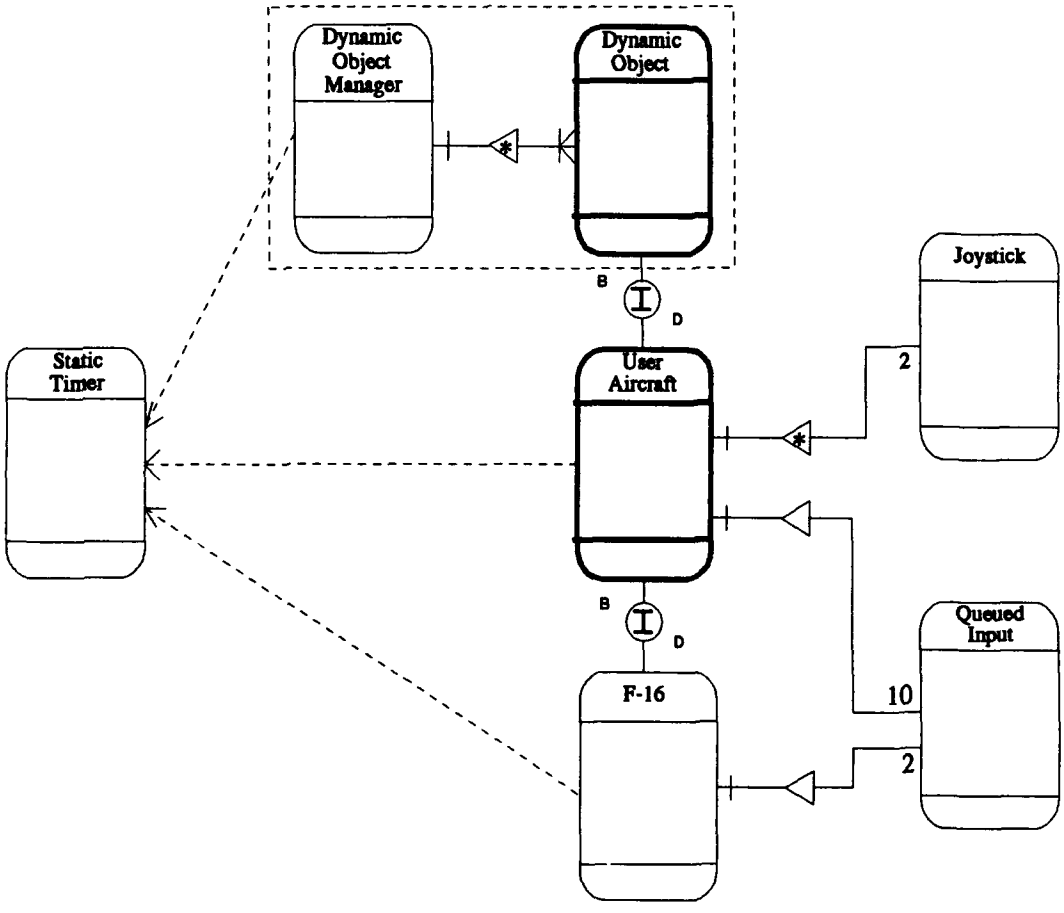


Figure C.3. Dynamic Object Class Hierarchy

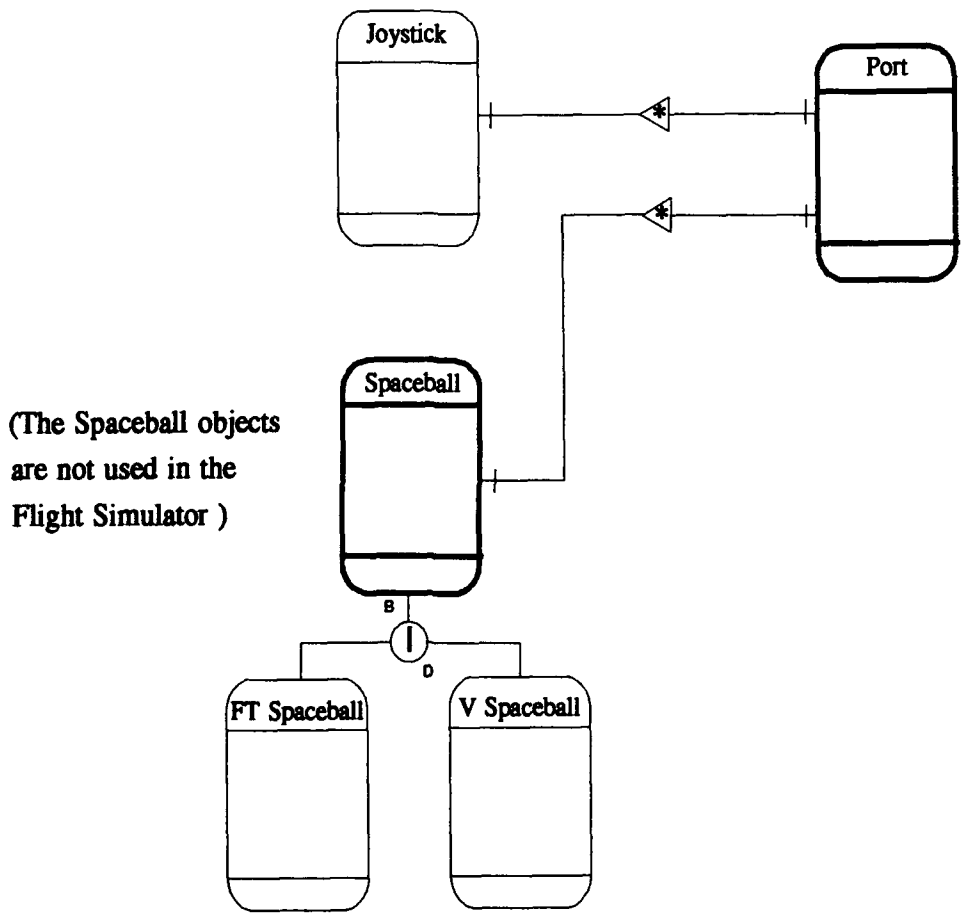


Figure C.4. Input Devices

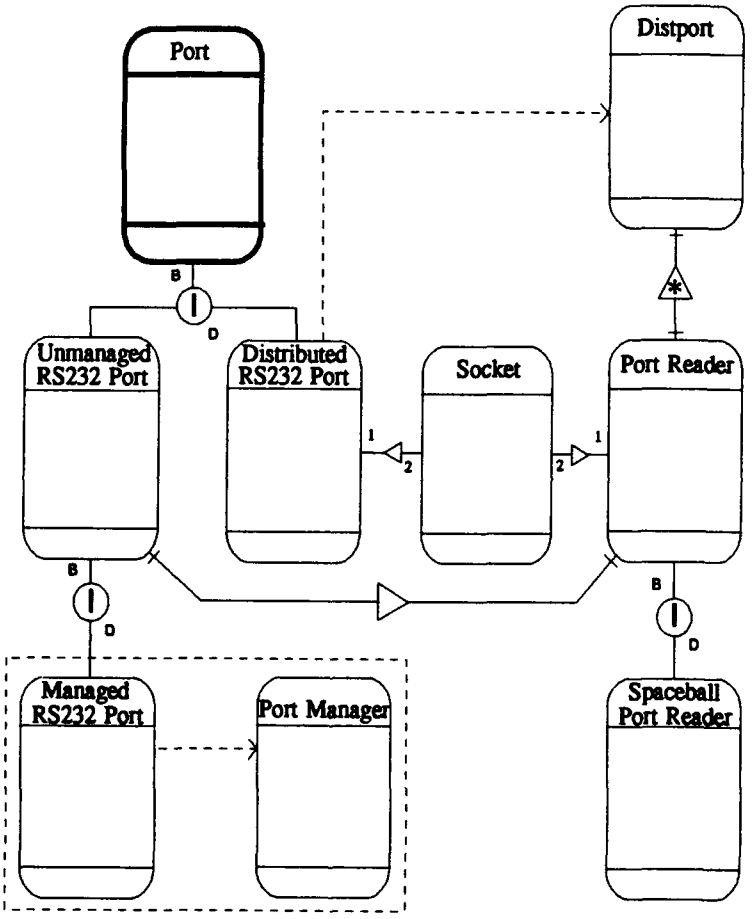
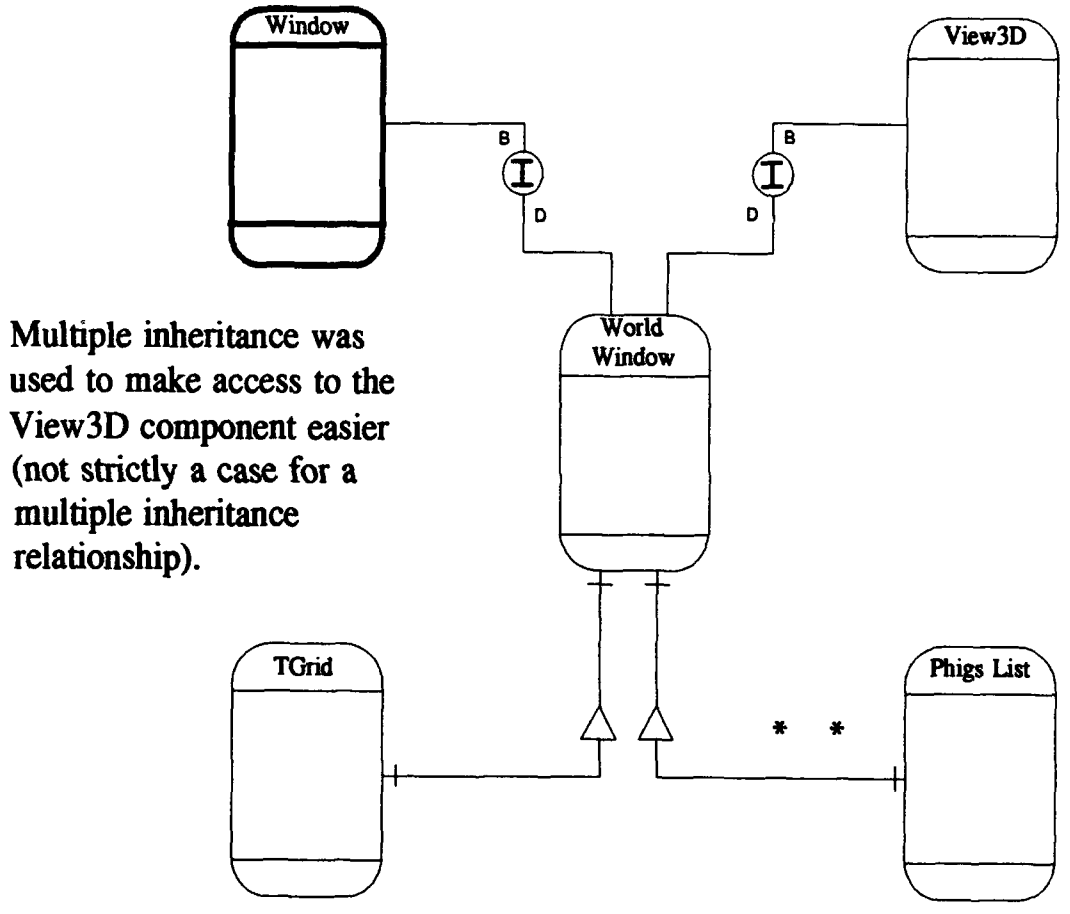


Figure C.5. RS232 Port Class Hierarchy



Multiple inheritance was used to make access to the View3D component easier (not strictly a case for a multiple inheritance relationship).

Figure C.6. World Window Class Composition (8, 38)

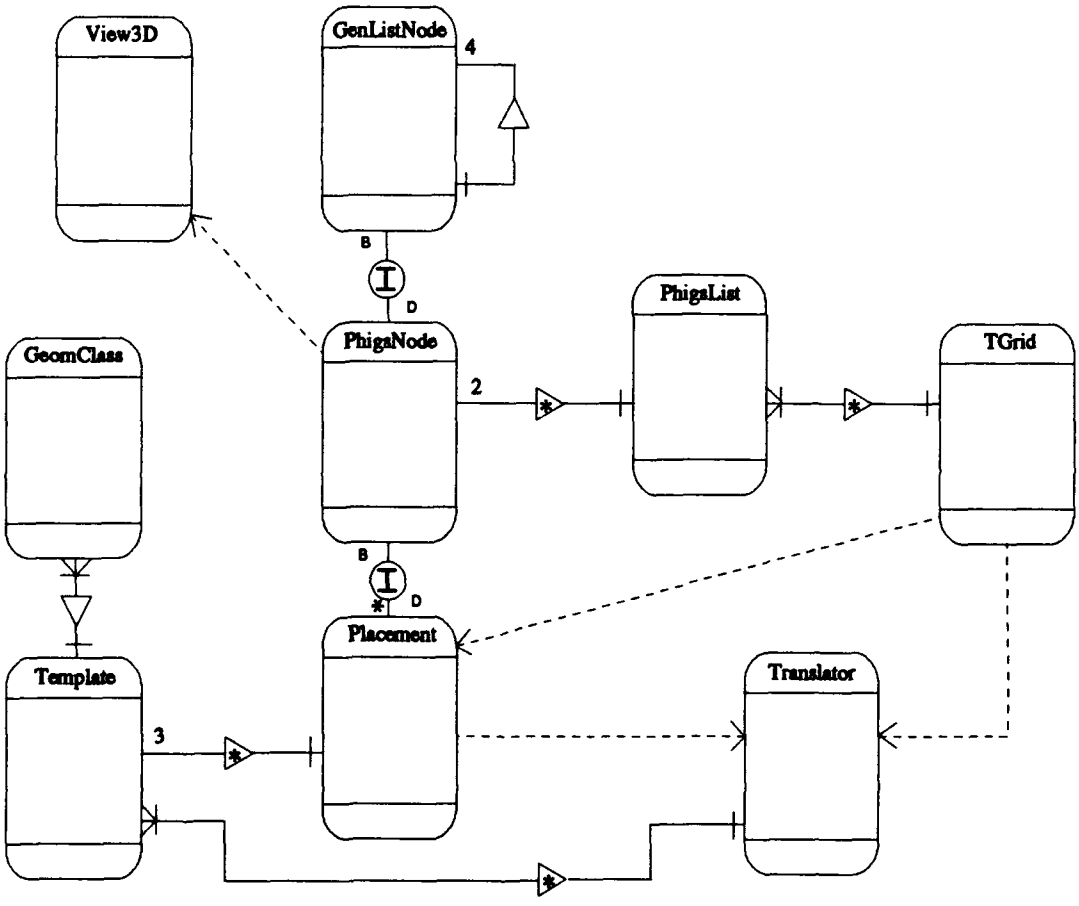


Figure C.7. Image Generation Classes (8, 38)

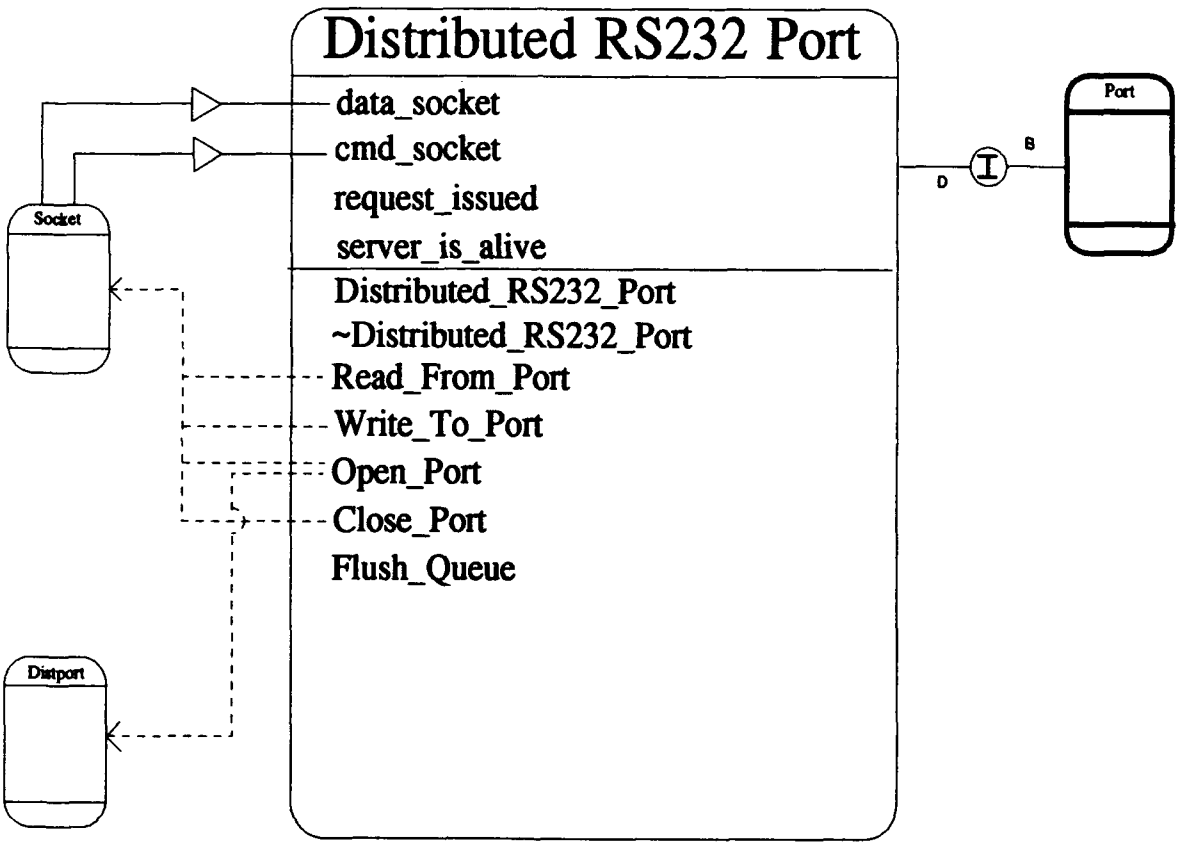


Figure C.8. Distributed RS232 Port Class

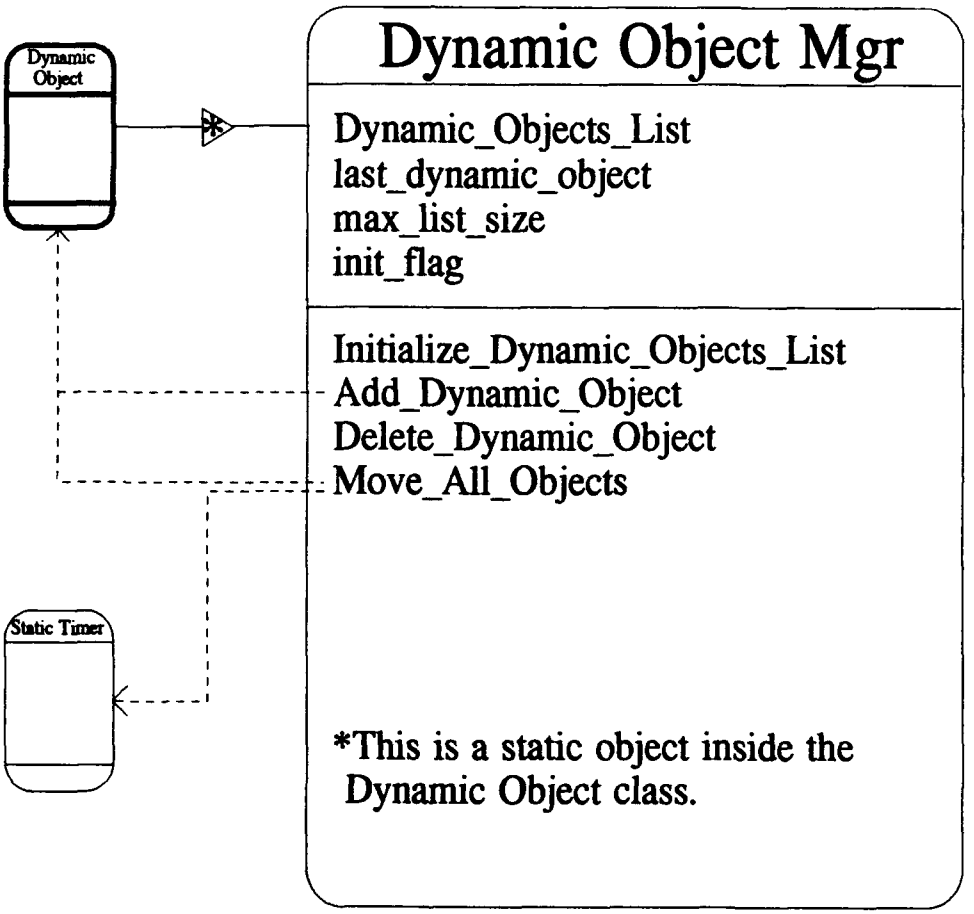


Figure C.9. Dynamic Objects Manager

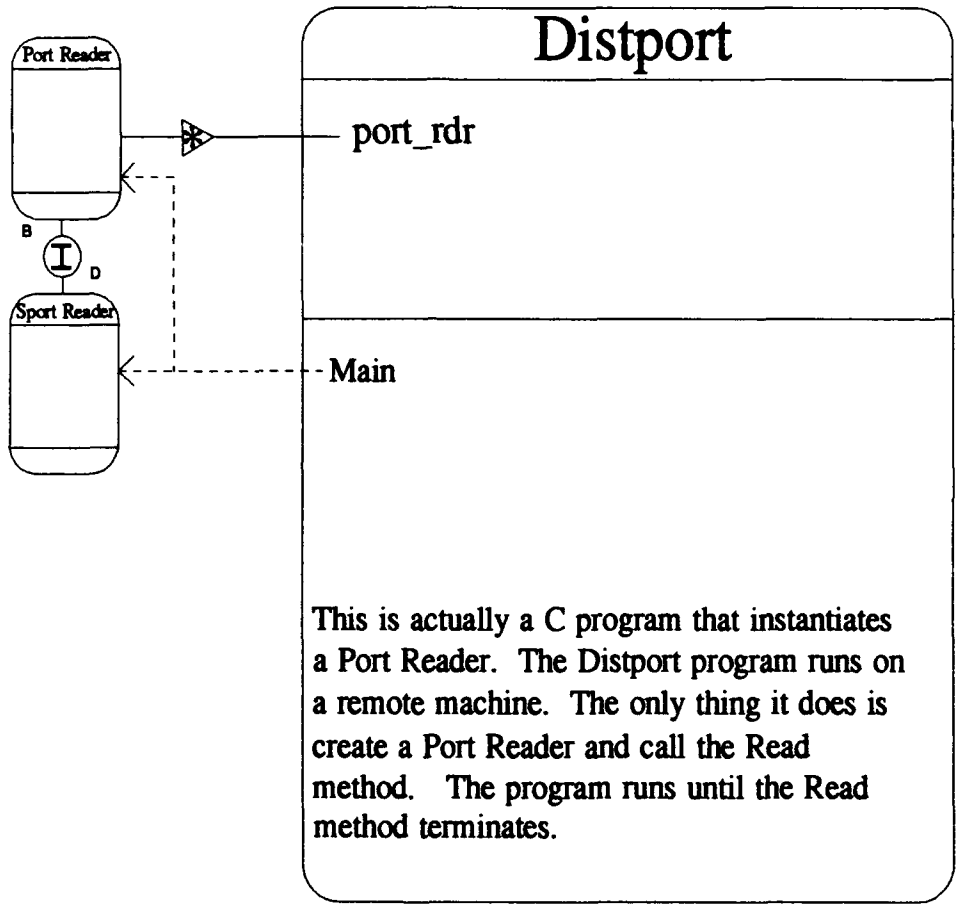


Figure C.10. Distport Module

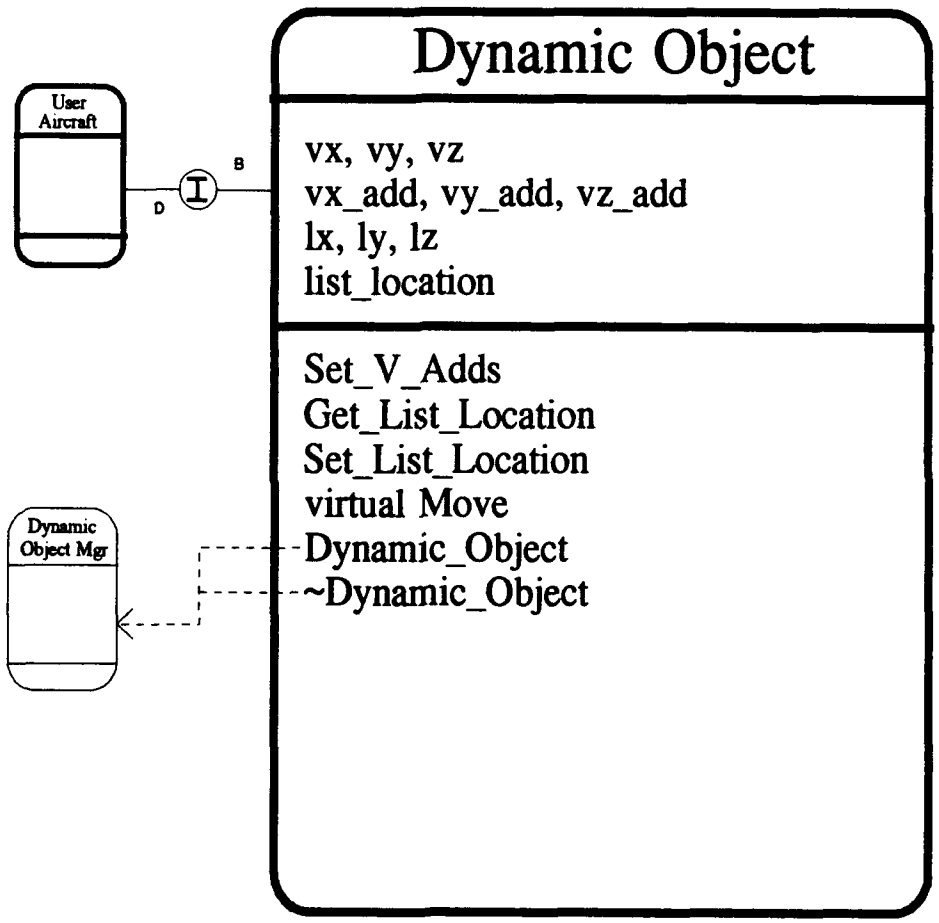


Figure C.11. Dynamic Object Class

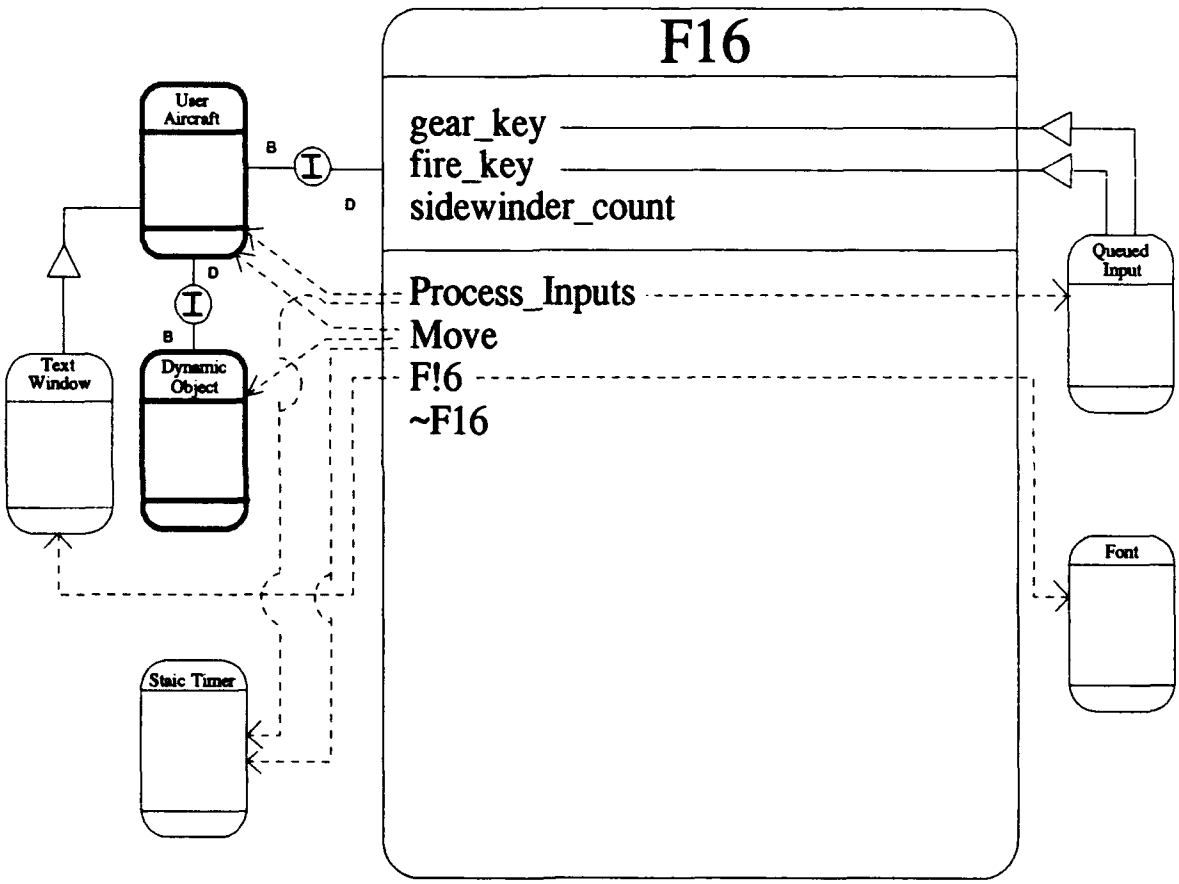


Figure C.12. F16 Class

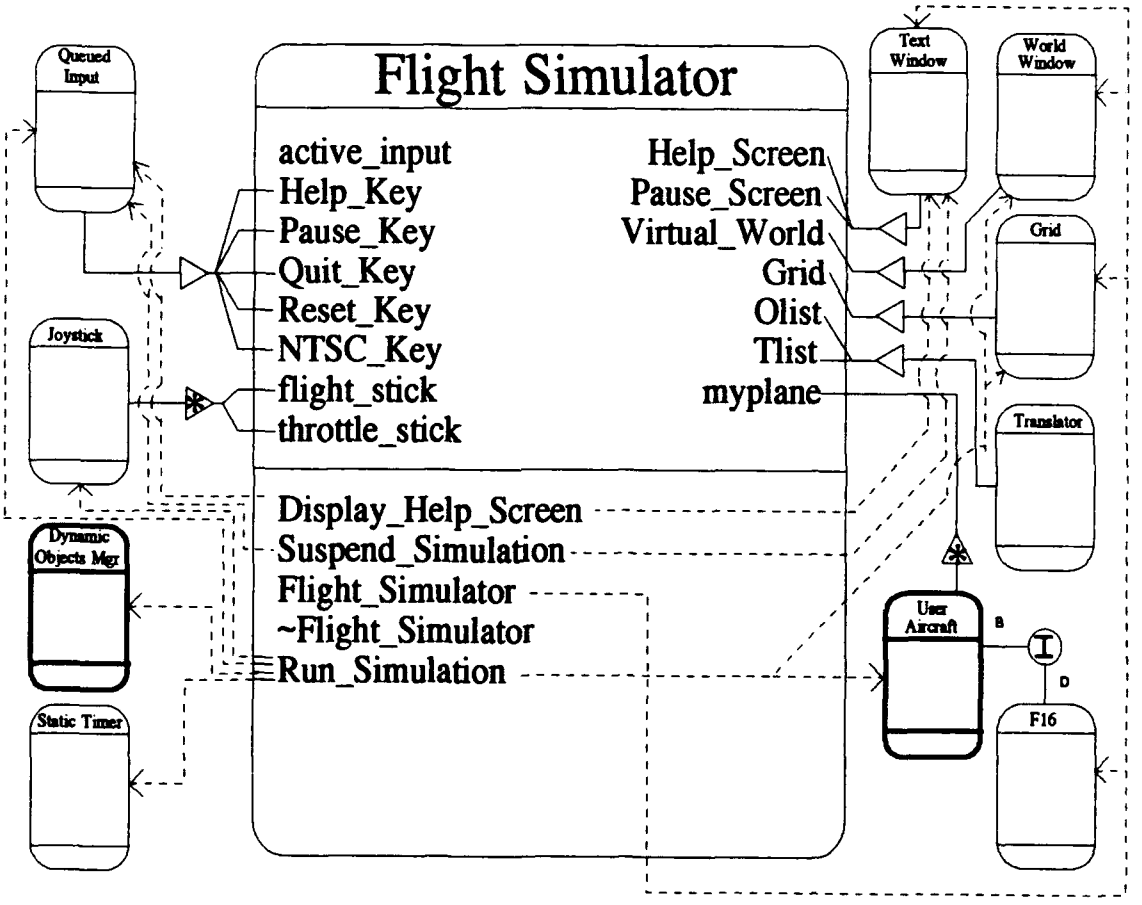


Figure C.13. Flight Simulator Class

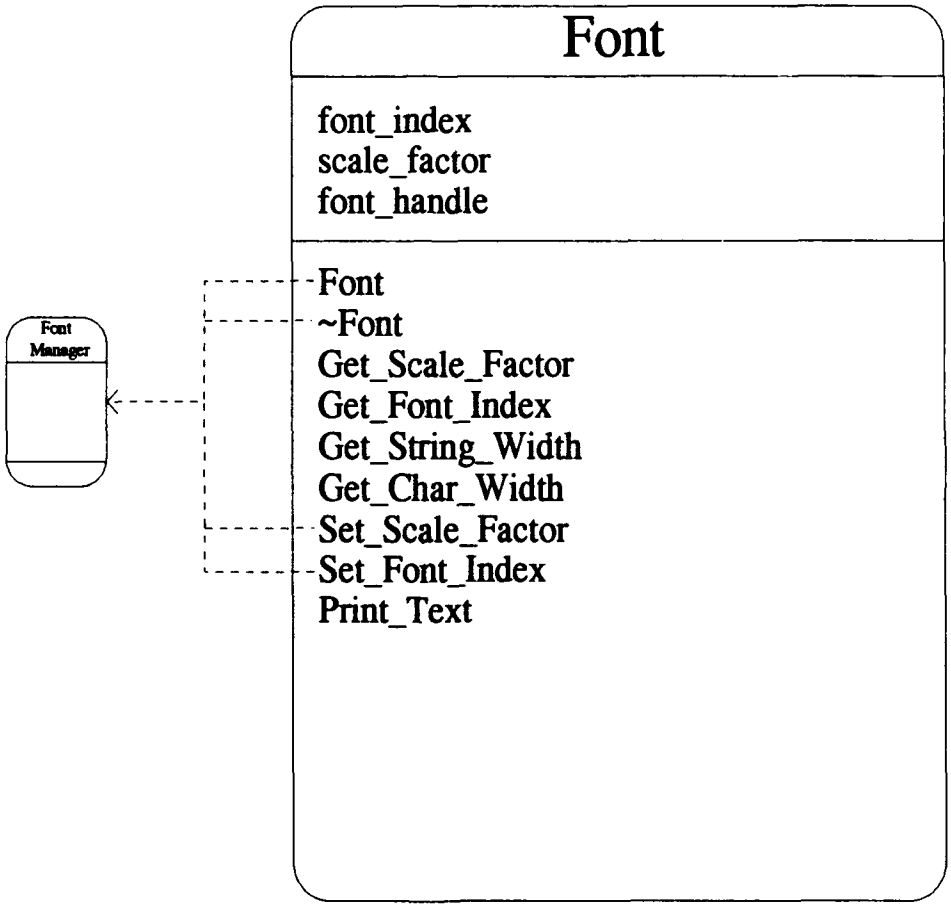


Figure C.14. Font Class

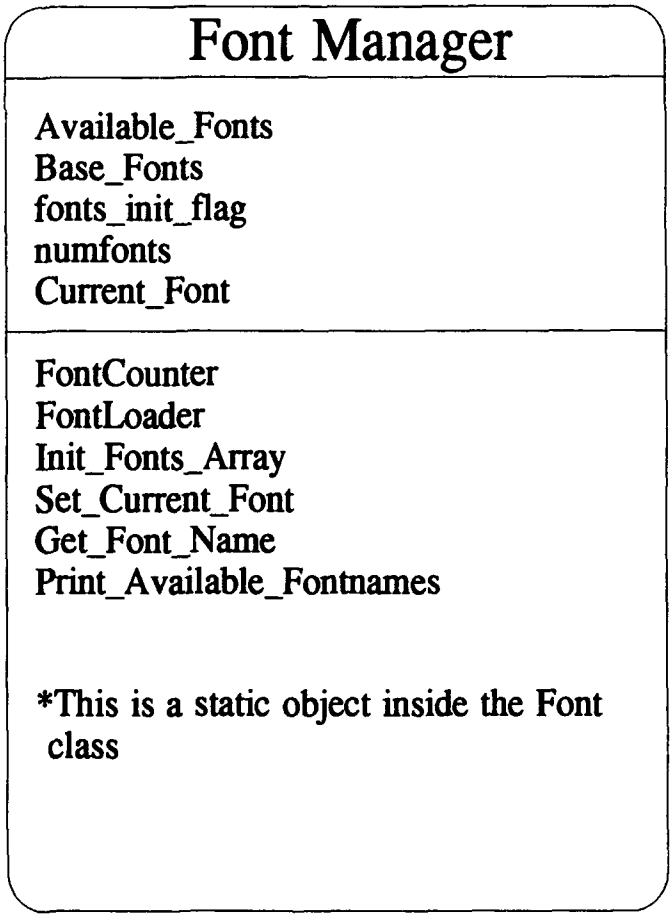


Figure C.15. Font Manager

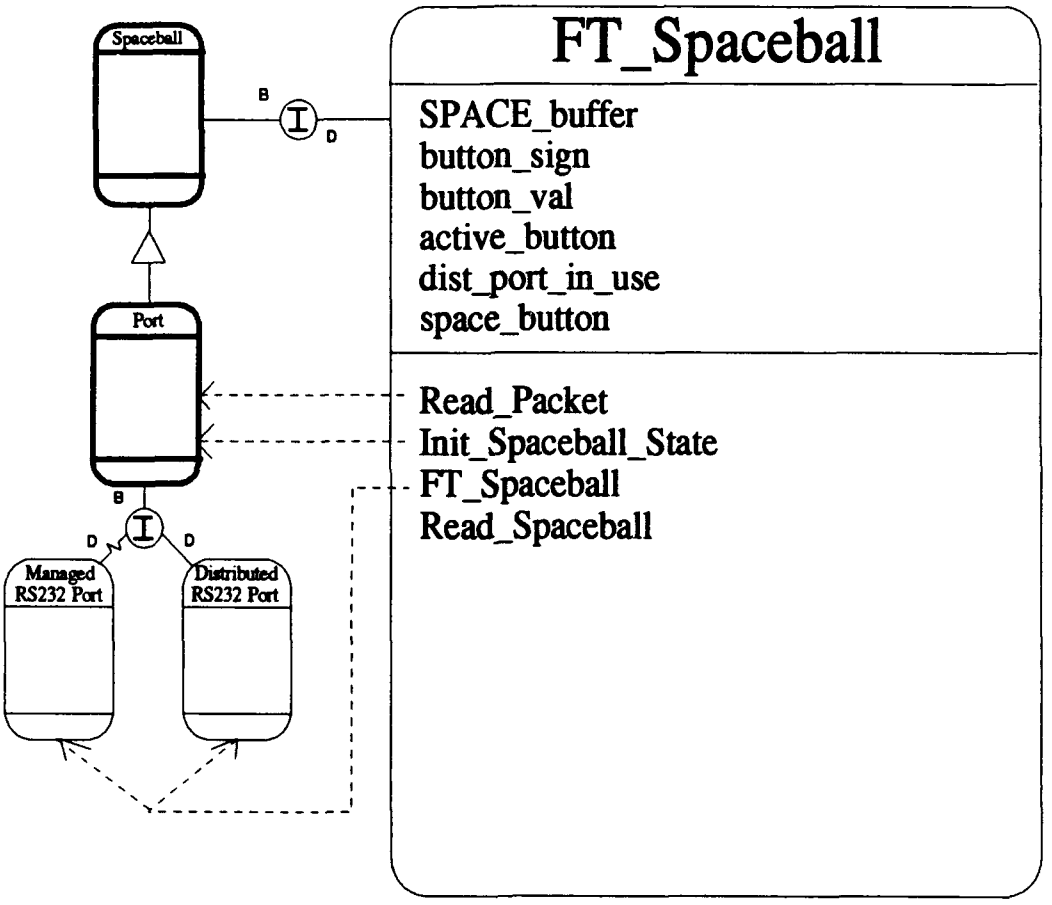


Figure C.16. Force Torque Spaceball Class

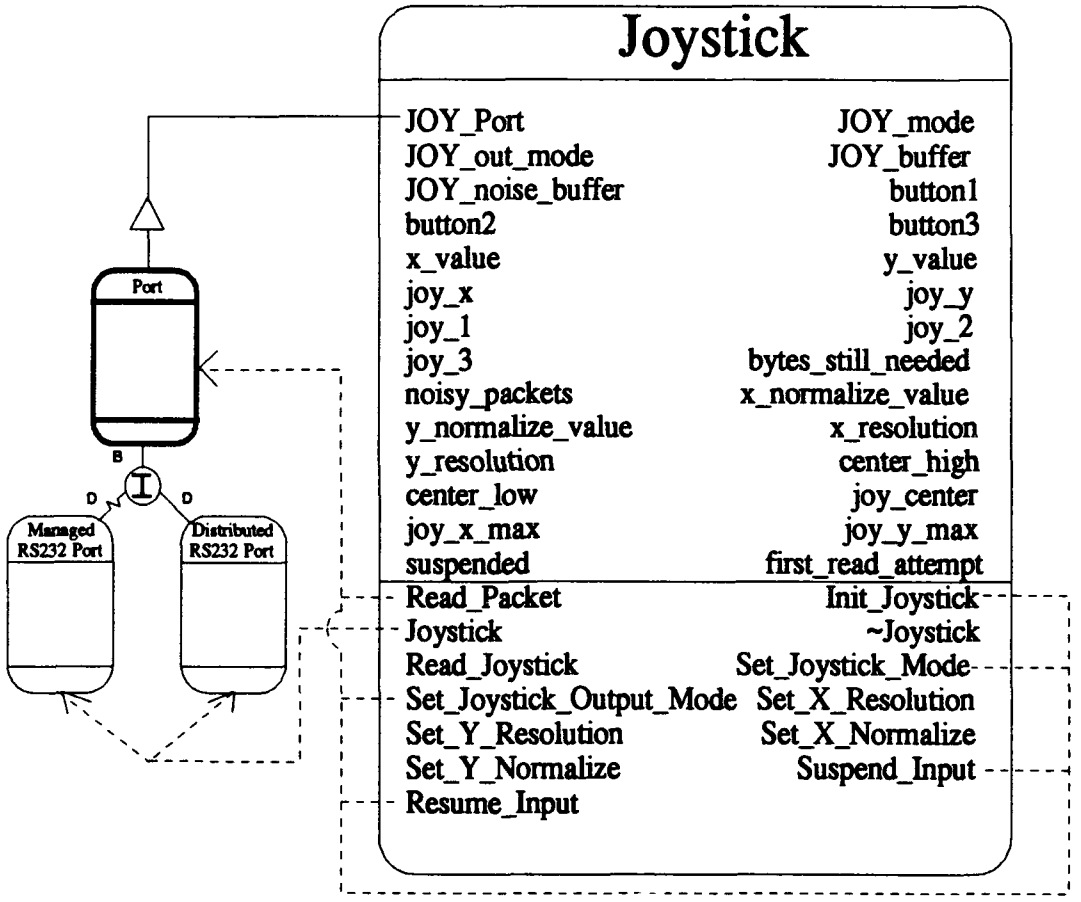


Figure C.17. Joystick Class

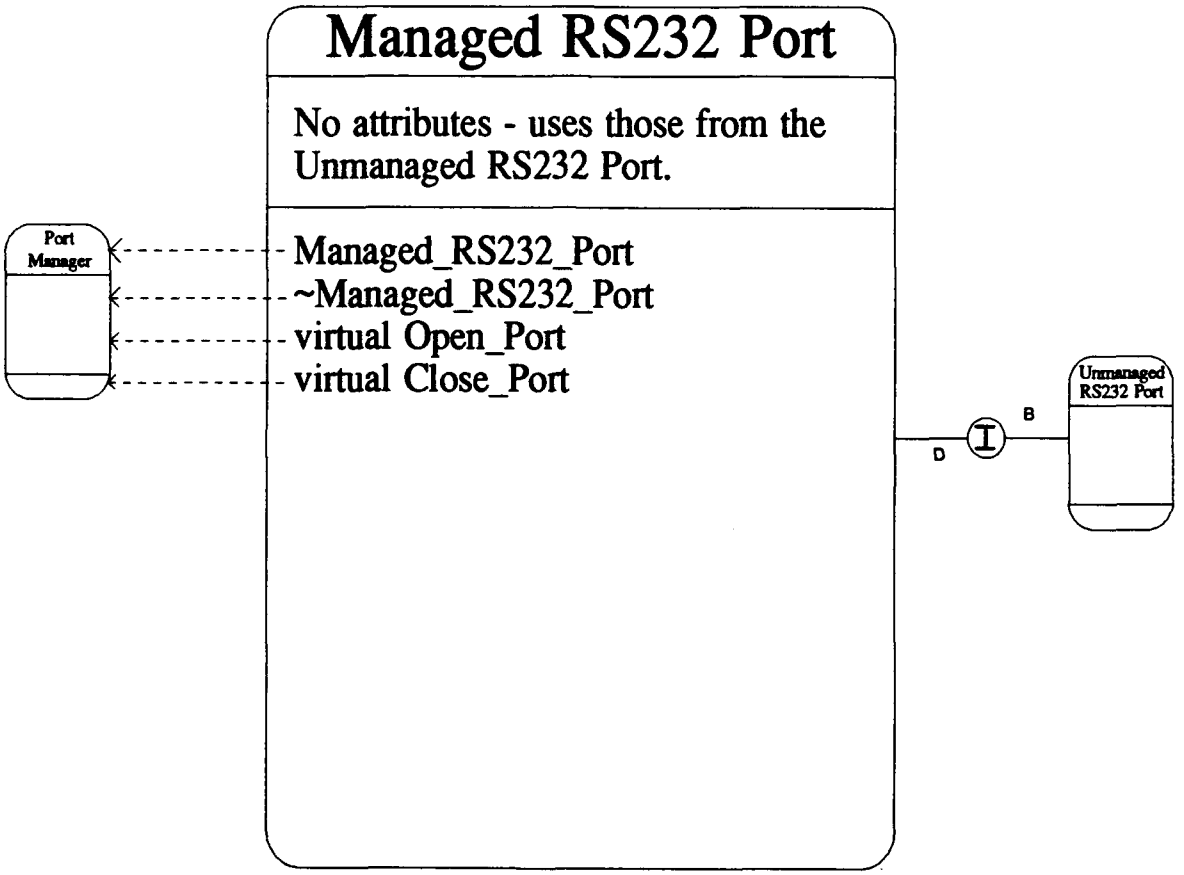


Figure C.18. Managed RS232 Port Class

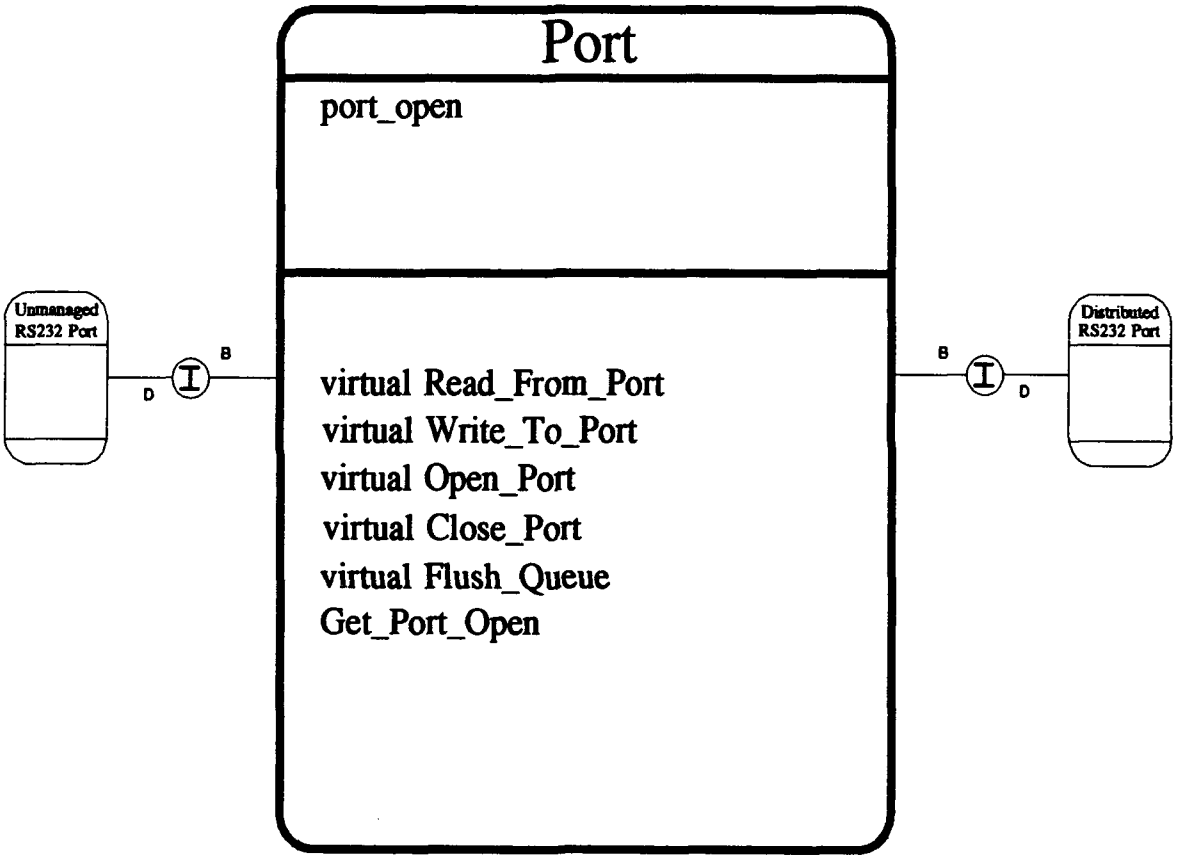


Figure C.19. RS232 Port Class

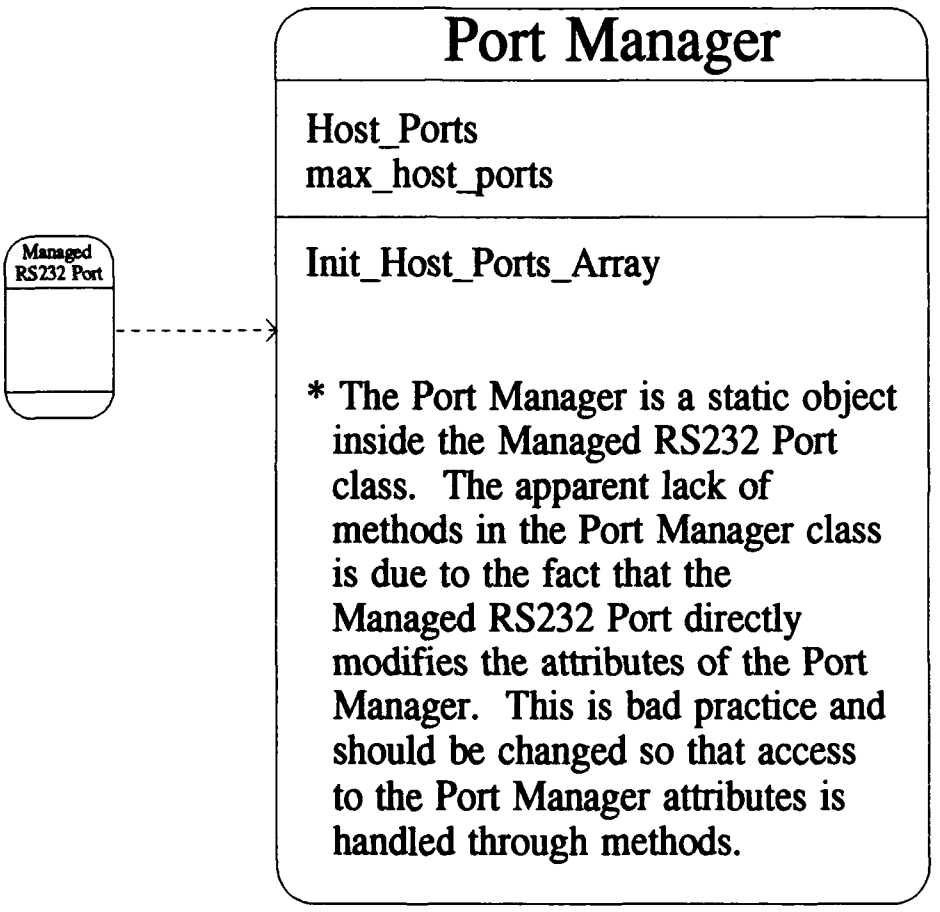


Figure C.20. Port Manager

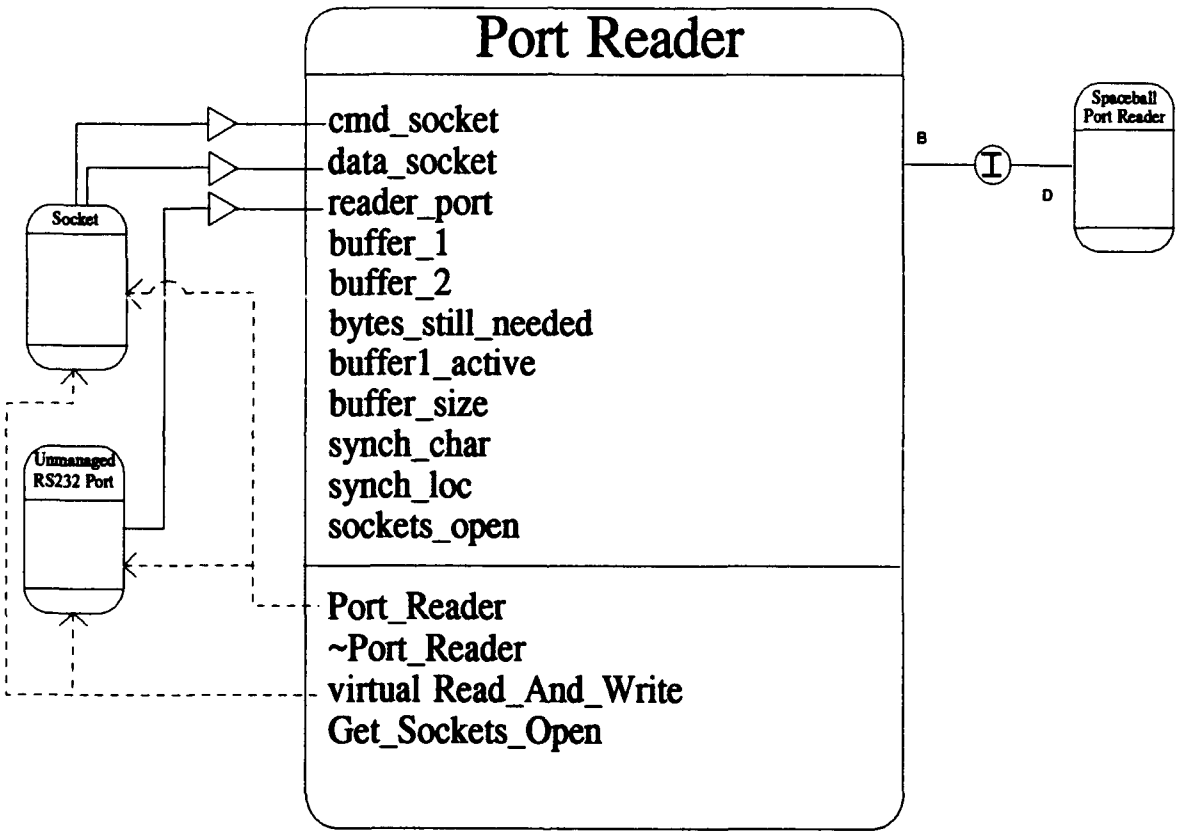


Figure C.21. Port Reader Class

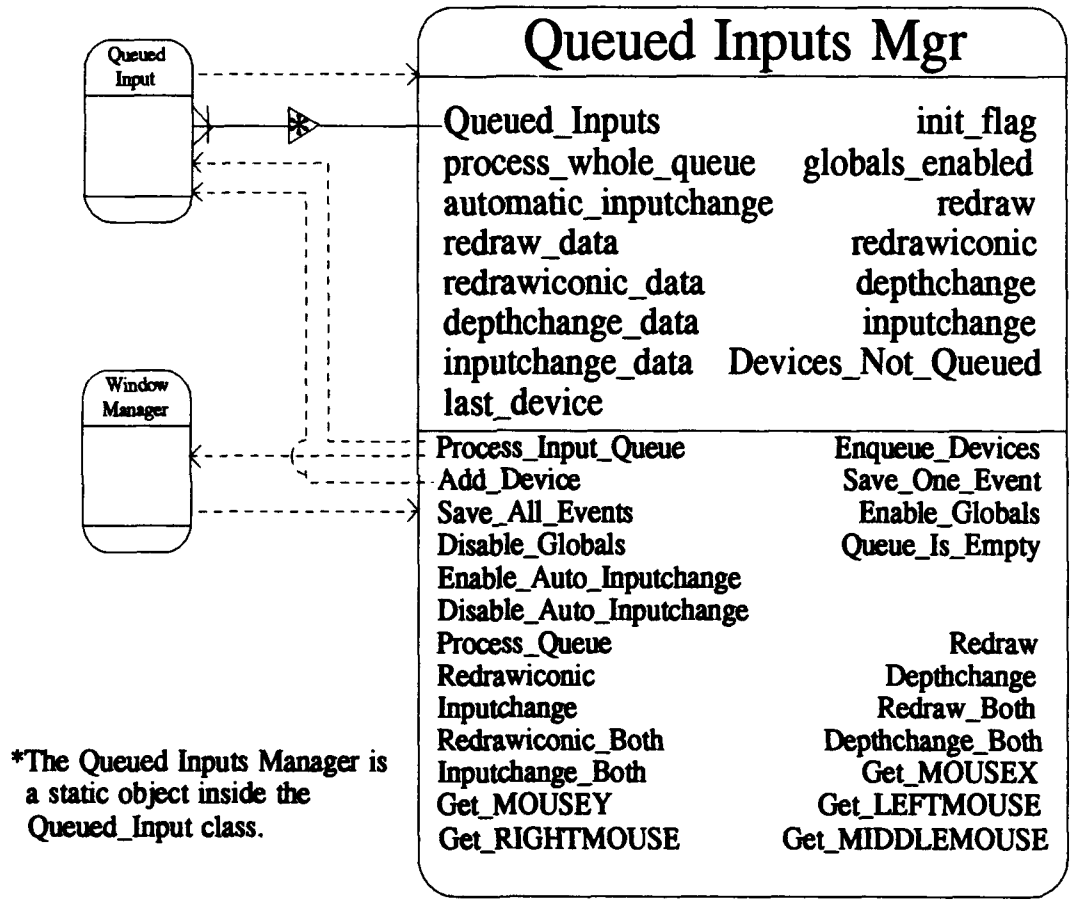


Figure C.22. Queued Inputs Manager

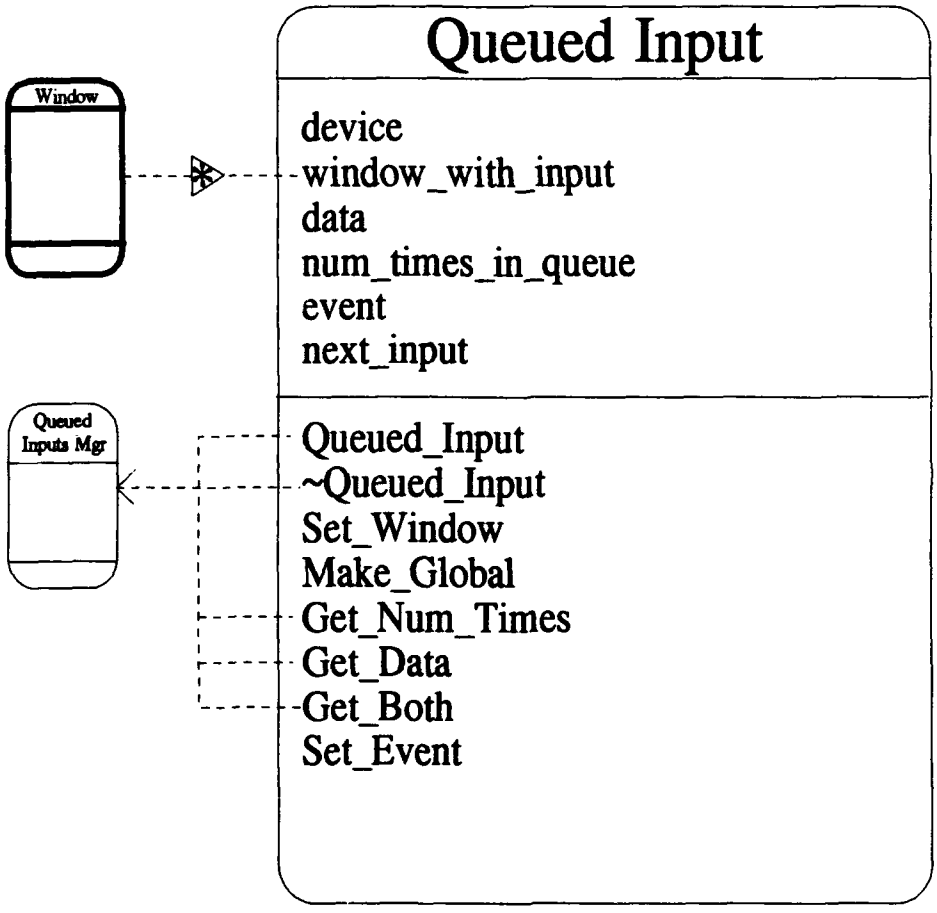


Figure C.23. Queued Input Class

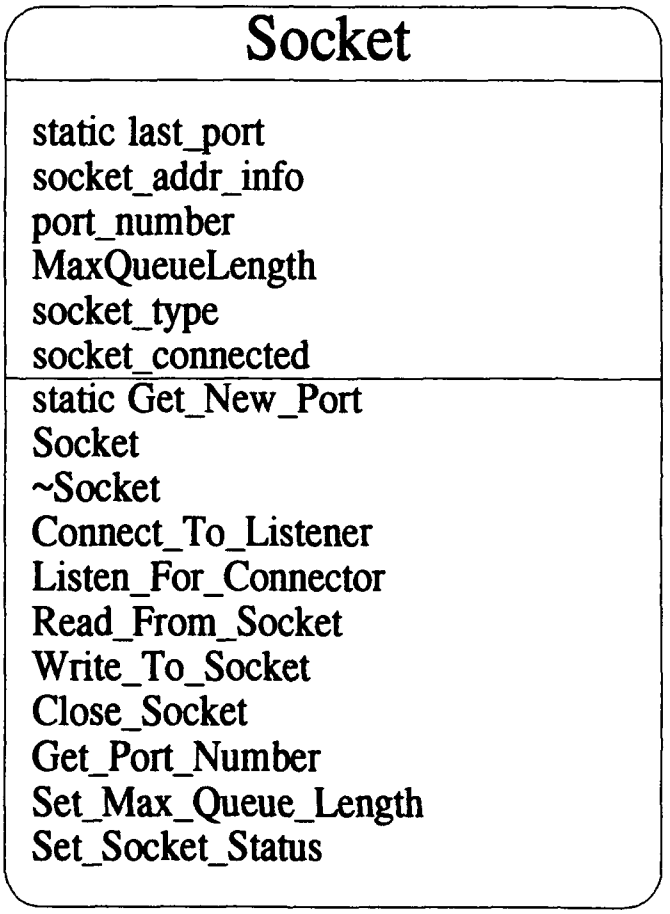


Figure C.24. Socket Class

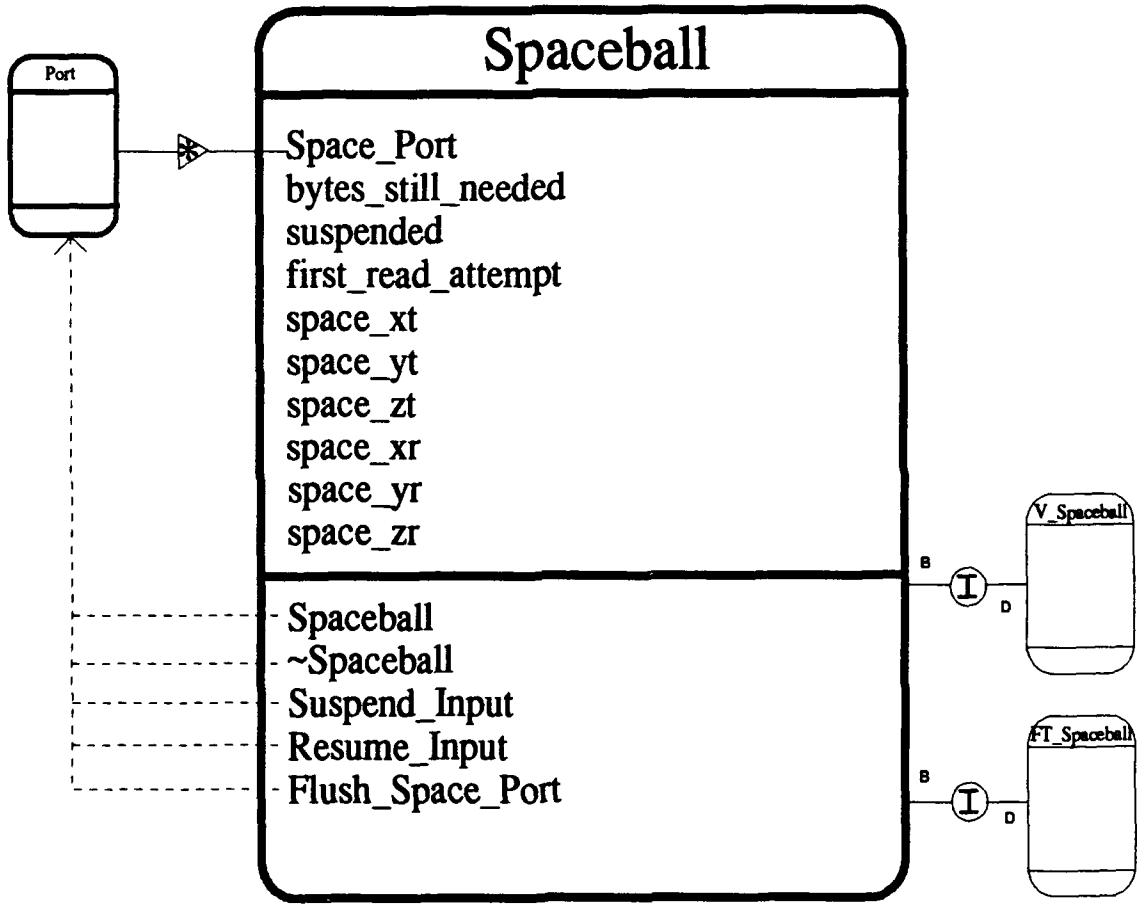


Figure C.25. Spaceball Class

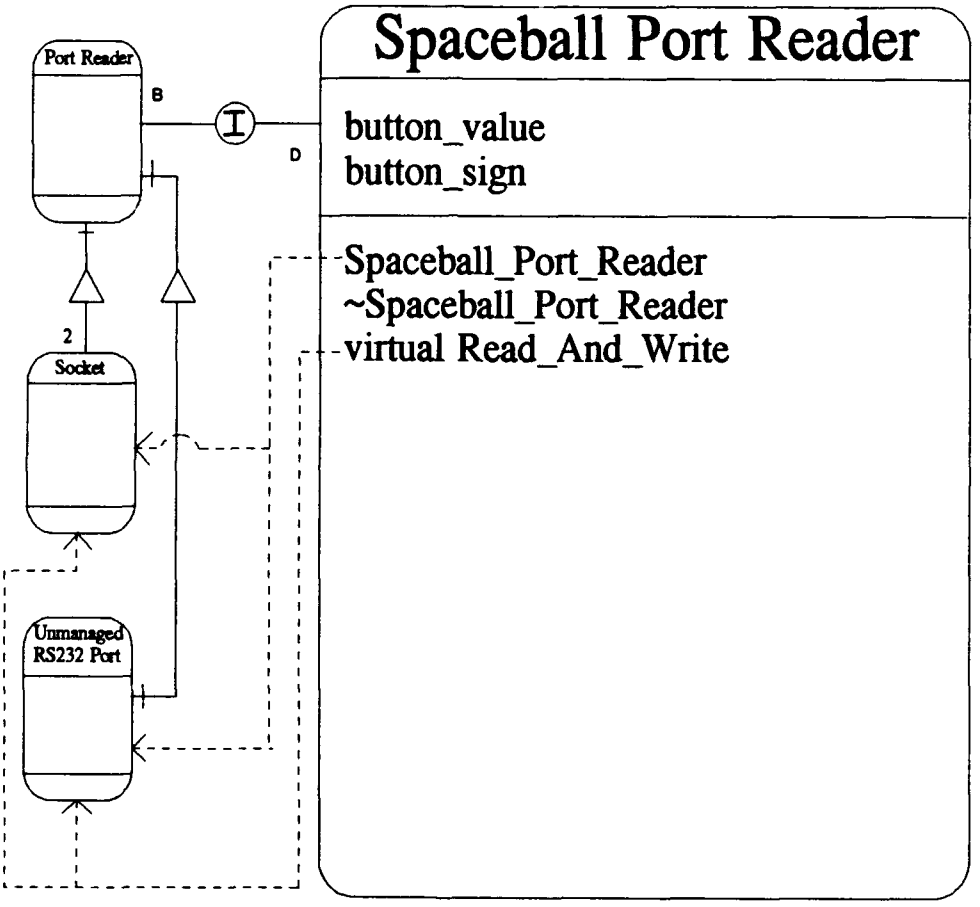


Figure C.26. Spaceball Port Reader Class

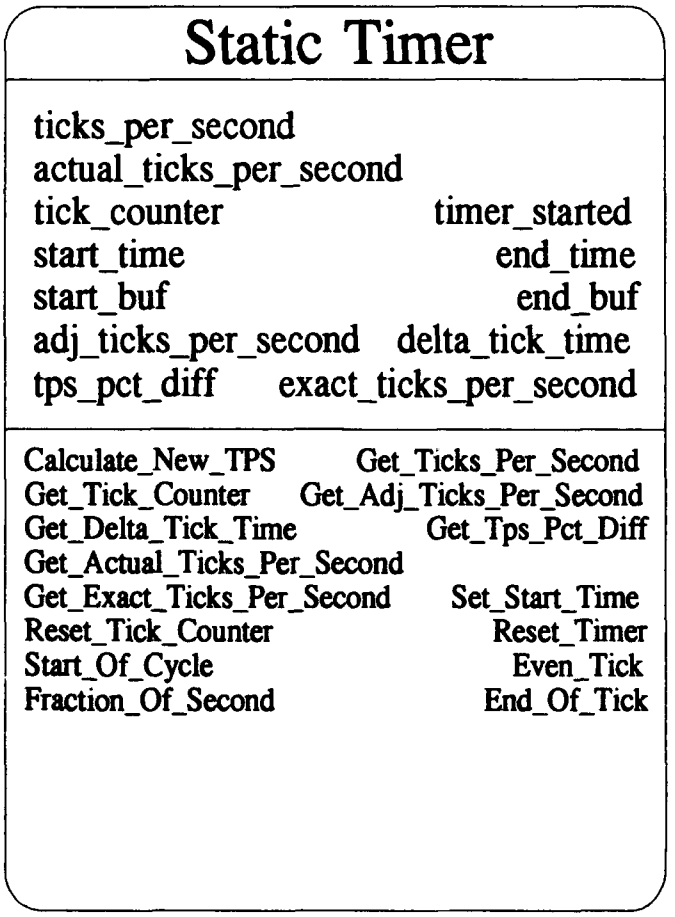


Figure C.27. Static Timer

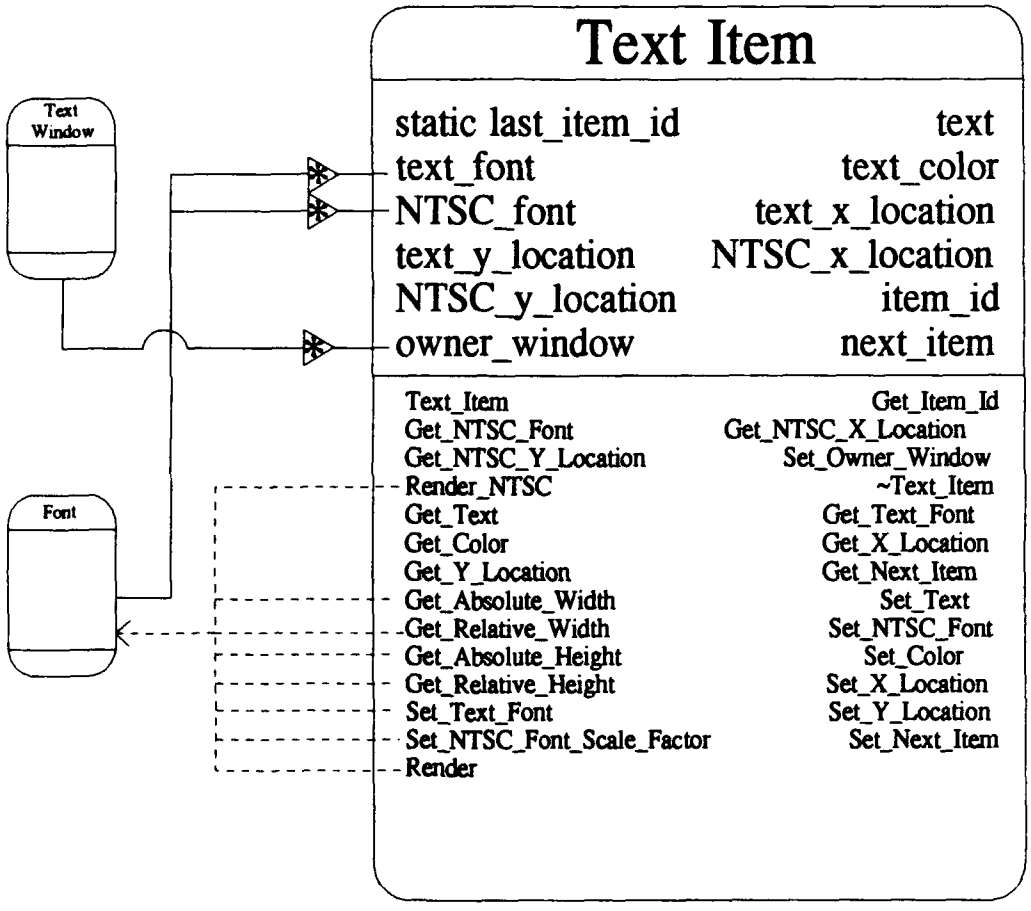


Figure C.28. Text Item Class

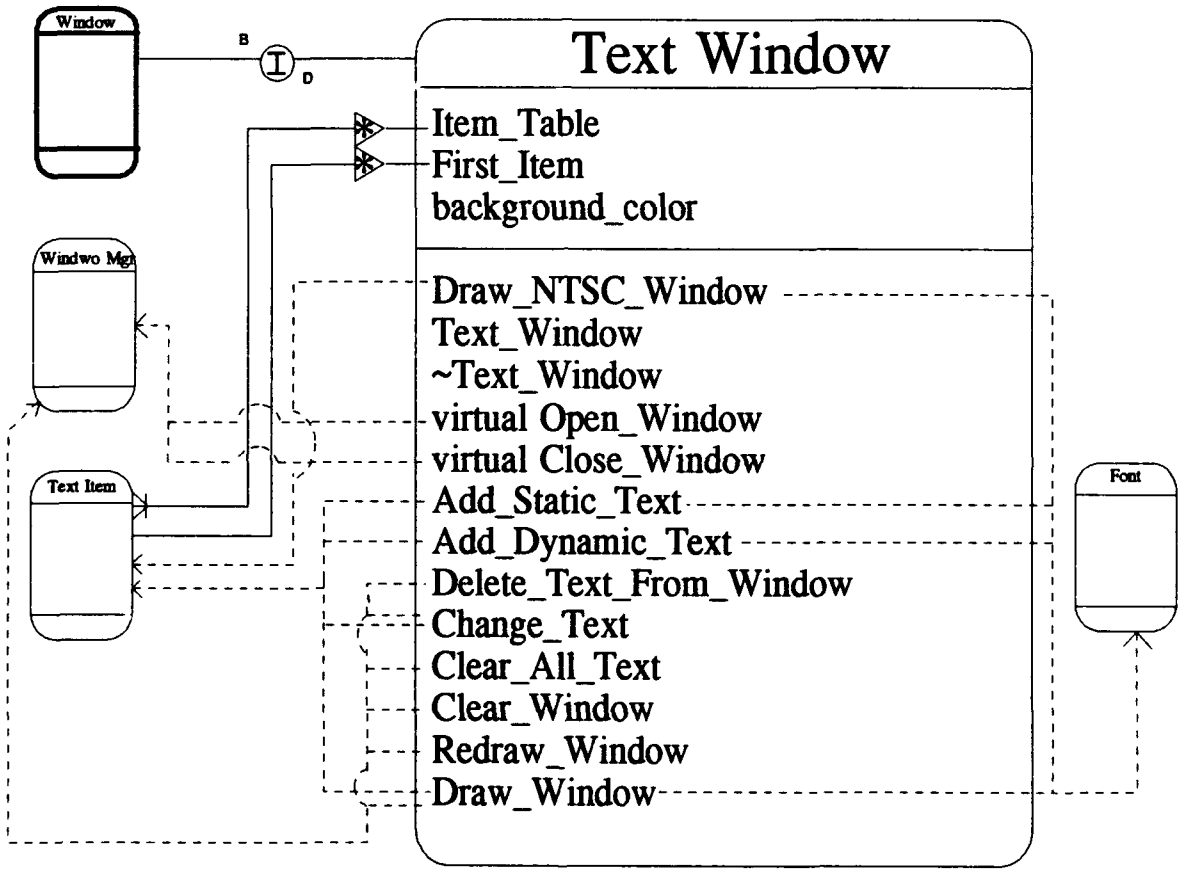


Figure C.29. Text Window Class

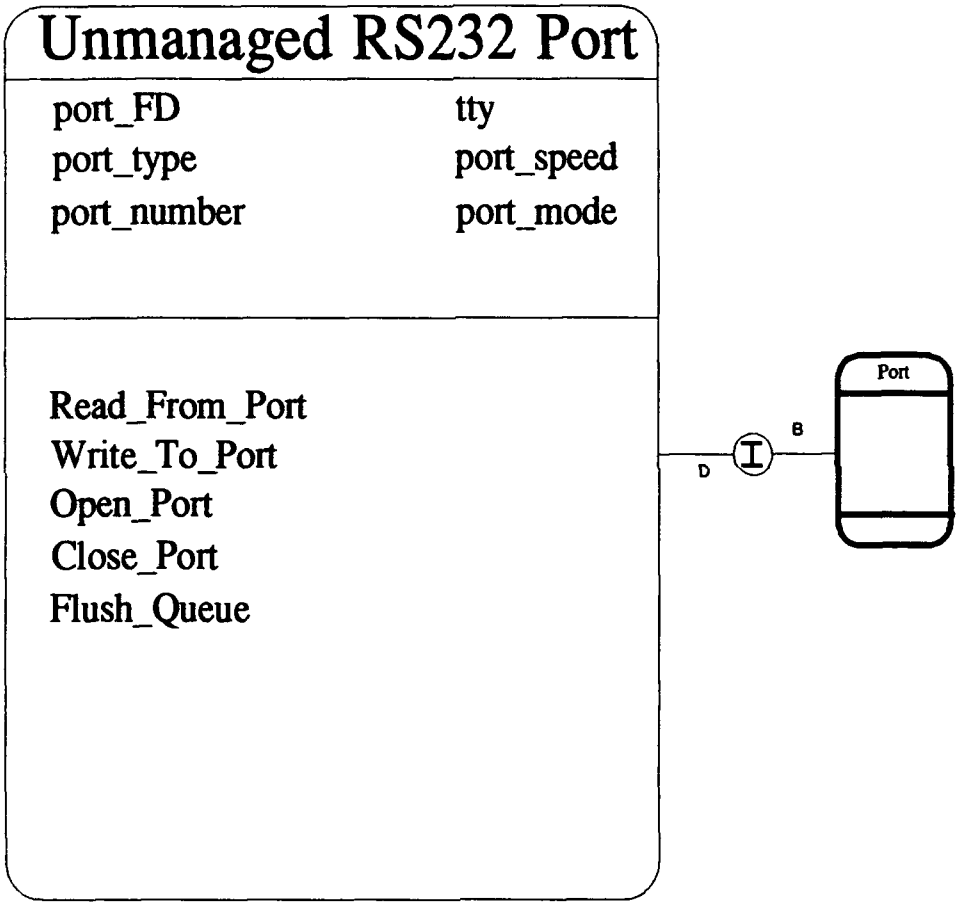


Figure C.30. Unmanaged RS232 Port Class

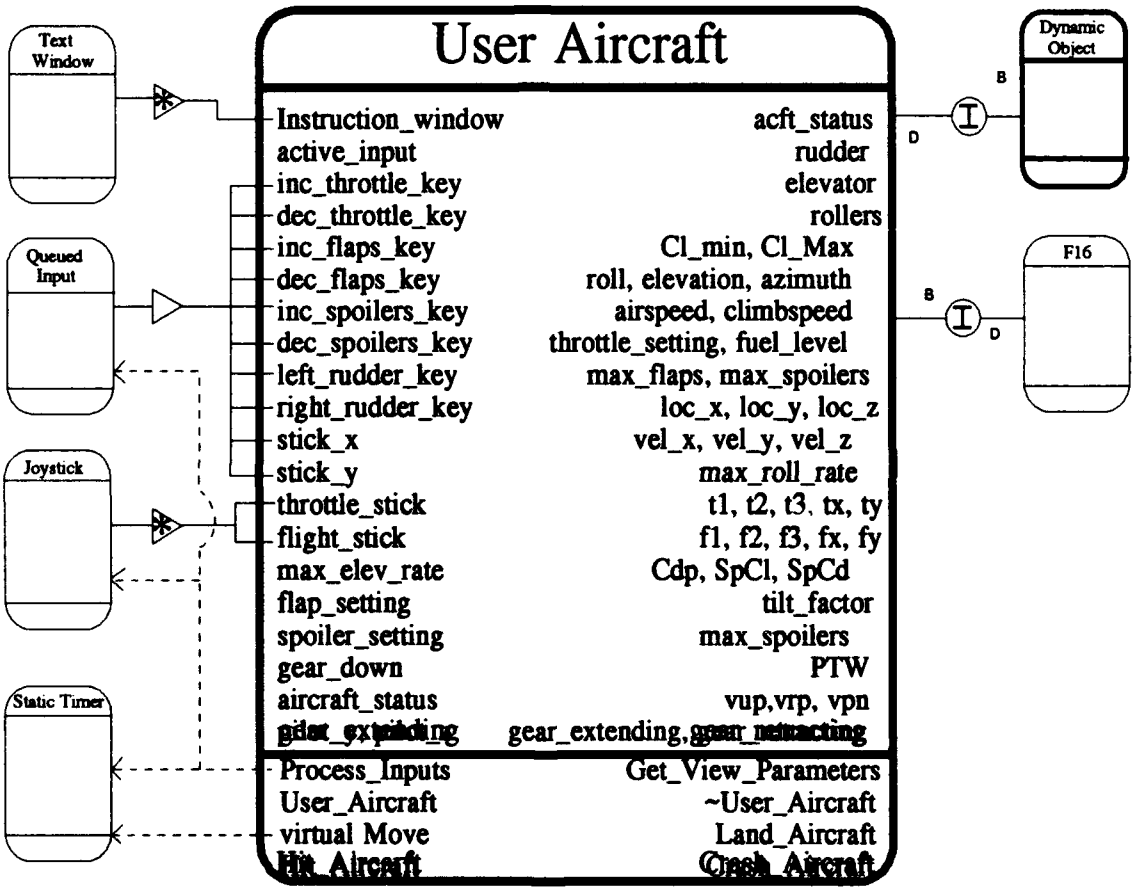


Figure C.31. User Aircraft Class

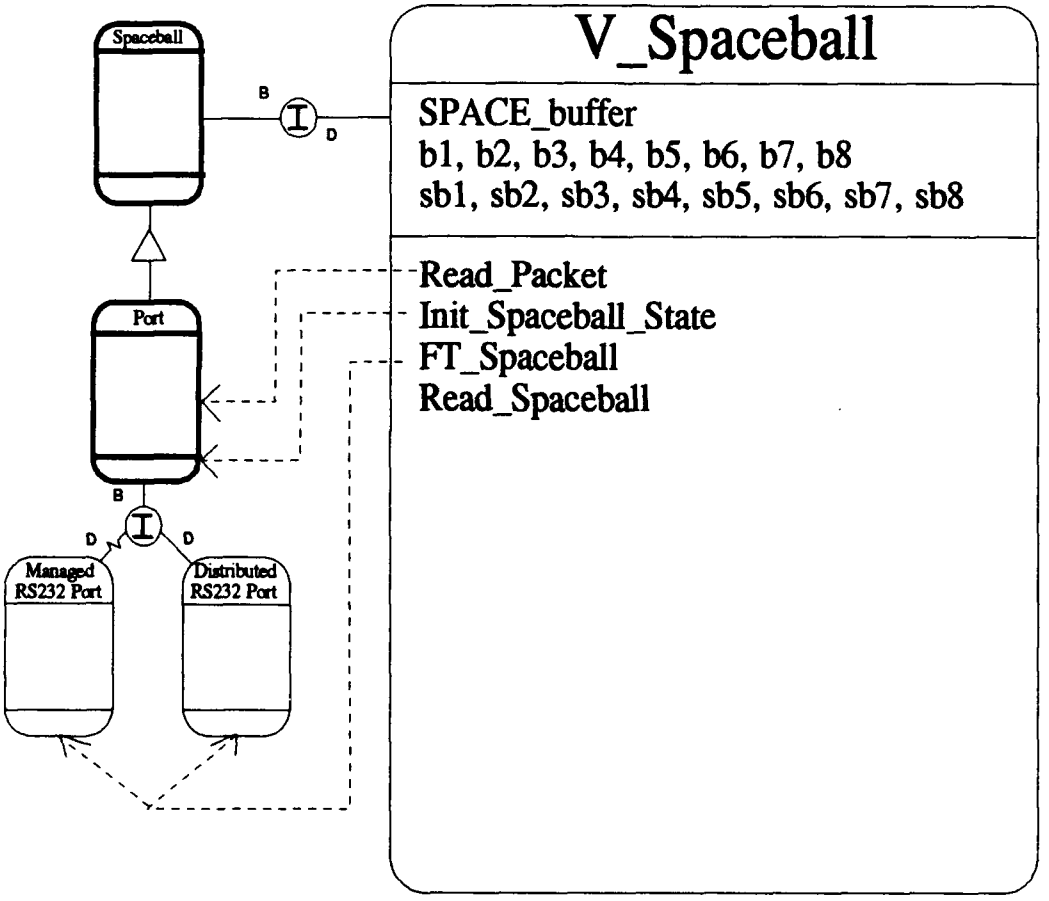


Figure C.32. Voltages Spaceball Class

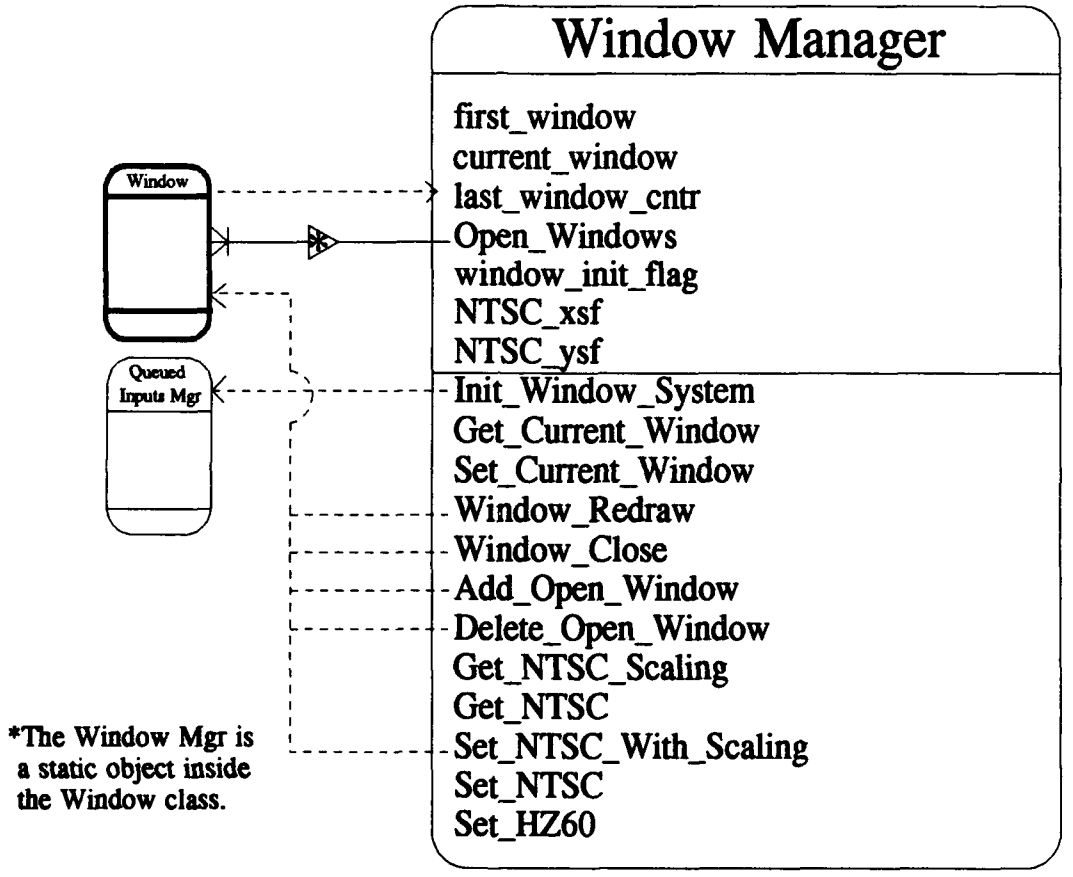


Figure C.33. Window Manager

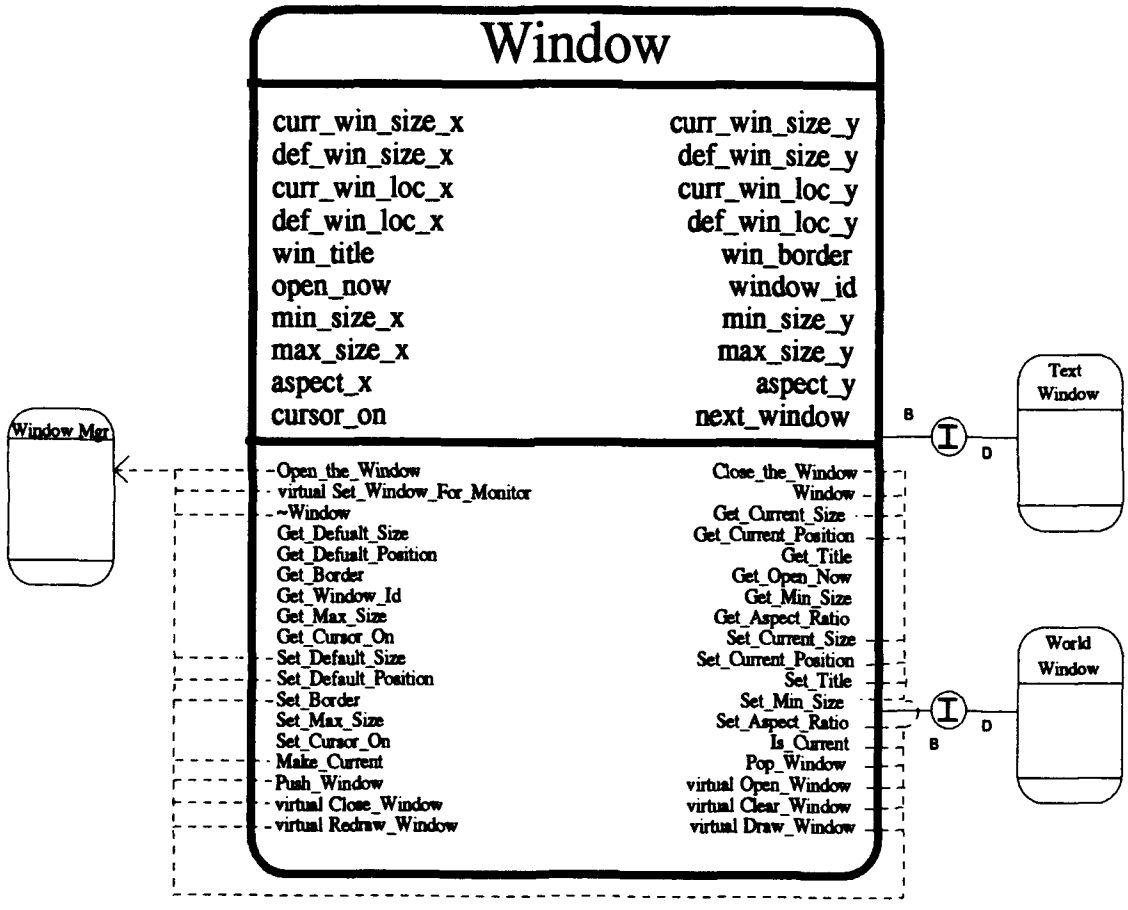


Figure C.34. Window Class

Appendix D. *Bibliography of Additional Object-Oriented Sources*

Anderson, Bruce and Sanjiv Gossain. "Software Reengineering using C++," *Proceedings of the Spring 1988 EUUG Conference*. 213-218. Buntingford, United Kingdom: European UNIX Systems User's Group, 1988.

Bensley, E. H. and others. *Distributed Object Oriented Programming.* Rome Air Development Center technical report RADC-TR-89-339, New Bedford MA; The Mitre Corporation, March 1990. (AD-A219 689).

Blake, Edwin. "Models and Actors," *Course Notes for course C-22 - Object and Constraint Paradigms for Graphics.* Course was given during SIGGRAPH 1991 - 18th International Conference On Computer Graphics and Interactive Techniques. III- 13 - III-34. ACM SIGGRAPH, August 1991.

Brück, Dag M. "Modelling of Control Systems with C++ and PHIGS," *Proceedings of the 1988 USENIX C++ Conference.* 183-192. Berkeley, CA: USENIX Association, 1988.

Cioch, Frank A. "The Impact of Object-Oriented Decomposition on Procedural Abstraction," *The Journal of Pascal, Ada and Modula-2 8:* 48-55 (May/June 1989).

Gibbs, Simon and others. "Class Management for Software Communities," *Communications of the ACM, 33:* 90-103 (September 1990).

Huber, Reiner K. and John M. Wozencraft. *On Distributed Wargaming in Operational C2 Systems Using Object-Oriented Programming Languages.* Naval Postgraduate School (NPS-74-85-001), June 1985 (AD-A157 331).

Laffra, Chris. "Object-Oriented Frameworks for Interaction and Graphics," *Course Notes for course C-22 - Object and Constraint Paradigms for Graphics. Course was given during SIGGRAPH 1991 - 18th International Conference On Computer Graphics and Interactive Techniques.* VI-17 - VI-32. ACM SIGGRAPH, August 1991.

Lee, Elgin. "The Journey of a Thousand Miles," *Computer Language*, 8: 44-54 (October 1991).

Mrdalj, Stevan. "Bibliography of Object-Oriented System Development," *ACM SIGSOFT Software Engineering Notes 15:* 60-63 (Oct 1990).

Rine, David C. "A Proposed Standard Set of Principles for Object-Oriented Development," *ACM SIGSOFT Software Engineering Notes, 16:* 43-49 (January 1991).

Rothenberg, Jeff. *Object-Oriented Simulation: Where Do We Go From Here?.* Prepared for the Defense Advanced Research Agency, report number N-3028-DARPA by the RAND Corporation, October 1989. (AD-A219 672)

Sakkinen, Markku. "Comments on the "Law of Demeter" and C++", *SIGPLAN Notices, 23:* 38-44 (December 1988).

Texel, Putnam "Object-Oriented Software Technology" A Primer for Managers, *Ada Strategies 4:* 3-8 (May 1990).

Weinberg, Randy and others. "Object-Oriented Systems Development," *Journal of Information Systems Management, 7:* 18-26 (Fall 1990).

Wirfs-Brock, Rebecca J. and Ralph E. Johnson. "Surveying Current Research in Object-Oriented Design," *Communications of the ACM, 33:* 104-124 (September 1990).

Wisskirchen, Peter. "Object-Oriented and Classical Approaches," *Course Notes for course C-22 - Object and Constraint Paradigms for Graphics.* Course was given during SIGGRAPH 1991 - 18th International Conference On Computer Graphics and Interactive Techniques. II-9 - II-22. ACM SIGGRAPH, August 1991.

Wybolt, Nicholas. "Experiences With C++ and Object-Oriented Software Development," *ACM SIGSOFT Software Engineering Notes, 15:* 31-39 (April 1990).

Appendix E. *Using C++ To Implement an Object-Oriented Design*

E.1 Introduction

The purpose of this appendix is to present guidelines for using different C++ constructs when implementing an object-oriented design. These suggestions are based upon the experiences gained from programming the prototype flight simulator. The recommendations are grouped with respect to the two low level design phases of object representation and providing object visibility. The following is a summary of this appendix:

I. Object Representation: decide how to implement the candidate objects in the design.
   - A. Implement the object with a class
   - B. Implement the object with a data type
   - C. Implement the object as a static object

II. Establishing Object Visibility: ensure that all data paths exist between objects that must communicate and specify the visibility of each class with respect to all others in the system.
   - A. Implementing required data paths
     1. Inheritance
     2. Composition
     3. Pointer to an object
     4. Parameter passing
     5. Static objects
   - B. Hiding data.
     1. C++ visibility mechanisms
     2. Hiding inheritance relationships
     3. Hiding information within an object

E.2 Object Representation

Object representation is an activity performed during low level design. The purpose of the object representation phase is to decide how the objects present in the high level design will be implemented. There are three ways to implement an object in C++: 1) implement the object as an object using the tools provided to define a class, 2) implement the object as a data type, and 3) implement the object as a static object.

The first decision to make is whether the object should be implemented as an object. This decision is based mainly upon whether the object provides any characteristic functions and if the object is supposed to hide any data. If the only functions that the candidate object provides are selectors and modifiers for each attribute, then it might be justified to implement the object as a data type.

If the candidate object lacks characteristic functions, then the selector and modifier methods must be examined in detail. A selector and modifier method must exist (in some fashion) for all attributes. The selectors and modifiers must be very cohesive. If either kind of method uses a different attribute than the attribute that is the target of the method, then the candidate object must be implemented as an object and not a type. In addition, the proposed selector and modifier methods must effectively grant unlimited access to the attribute(s) that they are connected with. There should also be no conditions upon how the attribute is selected or modified.

The determining factor is that the candidate object must not hide data nor provide any functionality other than to group attributes together in order to be implemented as a data type. The "Point" data type present in the flight simulator is an example of a data type used in the Flight Simulator. The only purpose it serves is to store x, y and z values. There is no other functionality associated with it (at

least not in this design).

The main advantage to using a data type versus a class to implement an object is that it is simpler to use a data type. However, implementing an object in this manner must be done with the previous discussion firmly in mind. The previously stated guidelines make it difficult to justify using a data type versus an object. The types of structures that do pass the tests will tend to be very small.

The disadvantage to implementing an object with a data type is that the programmer loses the benefits associated with implementing the candidate as an object. However, this may not be a disadvantage at all given the data type in question. If the candidate does meet all the tests to qualify as a data type, then the advantages provided by implementing the candidate as an object will not be needed. The point is to be convinced that an object can be properly implemented as a data type before actually implementing it.

Another disadvantage to using a data type versus a class is that it may complicate future modifications to the system. If future requirements dictate adding functionality to the data type, it will have to be switched to a class. Changing a data type to a class could be an arduous task. The programmer will not have the luxury of a clearly defined interface nor information hiding if this occurs. Use a class if there is any possibility that a data type will take on added functions in the future.

Implementing a candidate object through a class is the preferred way of implementing objects. This allows the programmer to take advantage of the benefits of object-oriented programming.

The third way in which to implement an object is by making a static object. Using static data members and methods was briefly discussed in section 4.4. The main drawback to using a static object is that there can only be one of them in existence in the system. This obviously limits their application.

Despite their somewhat limited applicability, static objects did find their way

into the Flight Simulator. They were most often used as "Manager" objects. These managers perform actions on behalf of specific classes of objects. Examples include the Window Manager, Queued Input Manager and the Font Manager objects.

There was only going to be one of these particular objects in the system. Implementing the managers as static objects within the definition of the class that they were most associated with served to keep the related pieces of code in one place. This made both of these highly coupled abstractions (the static object and the class) easier to use. The added coupling was justified by the added functionality that the static object would provide.

Using static data members and methods to implement an object lacks some of the information hiding that the class construct provides. Fortunately, this lack of information hiding only applies to the class in which the static object is defined. Any object of the class that the static object is defined within can access any methods and/or data of the static object. Information hiding with respect to the class in which the static object is defined must be enforced by the programmer. Directly accessing static data members should not be allowed unless it is done through static methods.

E.3 Providing Object Visibility

The results of the analysis and high level design specify that many different lines of communication must exist between various objects in the system. There are three ways in which this communication will be indicated in the design: 1) inheritance relationships, 2) composition relationships and 3) communication outside an inheritance or composition relationship. The two main activities of the low level design phase of providing object visibility center on implementing the required paths between objects that must communicate with each other and hiding data members and methods within objects.

E.3.1 Implementing Required Data Paths This phase of low level design may require changes to the design given the options that C++ offers for implementing data paths. The most common changes will be that attributes will be added or modified. Structural relationships should not change mainly because C++ can directly mirror the structural relationships in the system. The C++ programming language provides the following capabilities to implement lines of communication in a design: 1) class inheritance (including multiple inheritance), 2) the ability to declare that an attribute be an object of another class (composition) 3) the ability to use a pointer to an object, 4) the capability to pass objects (or object pointers) as parameters to methods, 5) the ability to use static data members and static methods.

In general, inheritance relationships should only be used where they are indicated in the design. The product of the design is a model of the real world. Using inheritance simply to make things easier to implement clouds the design. This conflicts with one of the benefits of using object-oriented design in the first place — making the program easier to understand. Using inheritance without any purpose will make subsequent maintenance of the program more difficult.

The urge to use inheritance to provide extra visibility to an object occurs when a programmer is faced with a composition relationship in which a component object must be accessed frequently by other objects in the system. If the proper relationship were implemented, then outside objects would need to call a selector to get to the desired component object and then make method calls from it. If the component object and the object that contained it were related by inheritance, an outside object would be able to call the desired method of the component object directly. This would eliminate one method call and makes access to the component easier. Although it should be avoided, if inheritance is used in this manner, it should be justified and documented in the code.

Like inheritance, the composition relationship should be used where it is indicated by the analysis and high level design. Again, this concerns remaining faithful

to the structure indicated by the previous phases. However, it may be necessary to add a composition relationship that is not in the design.

The decision to add a new component relationship is driven mainly by the fact that every entity in the system must be a part of another. With the exception of making static objects, there is no way to make autonomous objects. There must be some point in the program that will reserve space in computer memory for the object and keep the space until it is not needed any longer. Adding a component relationship is sometimes the only option available in order to establish and maintain the existence of an object in the system.

The third way to implement a communication path is to use a pointer to an object as a component of a class. Using a pointer to an object allows the programmer more flexibility as opposed to using the actual object as a component. Using pointers to objects enables polymorphism, allows the container object access to an object that was possibly instantiated in another object, and it allows the programmer more flexibility to specify how a component object should be instantiated.

The C++ language implements polymorphism solely through pointers to objects. To use polymorphism, the programmer declares a pointer to the base class as a component. Given the pointer to the base class, the programmer could instantiate an object of any derived class of the base class. After doing this, all the programmer would have to do to make polymorphic calls was to make method calls using the pointer to the base class.

Using a pointer to an object also allows the object to be shared among other objects. An example from the Flight Simulator are the pointers to Joystick objects defined in the User Aircraft and F16 classes. They were defined as pointers because the Flight Simulator object was the one that actually told the computer to reserve the space for them. There could only be two Joystick objects in the simulation at a time. Using pointers allowed them to be shared among different objects.

Using pointers to objects also allows the programmer more flexibility in instantiating the object. If the actual object was declared, then there is only one place available to instantiate it — in the parameter specification of the constructor of the container object. If more flexibility is desired, then a pointer to the object must be used. The object could then be instantiated within the code of the constructor in any way the programmer wished.

There may be situations in which the programmer may not wish to actually instantiate the object at all. This was also exemplified in the Flight Simulator object. If the user does not wish to use the joysticks, then the Flight Simulator will not instantiate them. It simply passes a null pointer down to classes that may have wanted access.

Using pointers to objects also has some disadvantages. If an object is granted access to a pointer, it could inadvertently delete the storage for it when it really didn't "own" it. Conversely the object that does "own" the object could delete it when other objects still had the pointer to the now defunct object. Allowing access to a component pointer (eg. through a selector) should be avoided. The safest thing to do is make a copy of the object and then pass the pointer to the copy.

A fourth way that C++ lets a programmer establish a data path is through the parameters of methods. The main problem with this approach is determining what other object has visibility to all of the objects involved. Some object will have to be responsible for performing the method call. That object must have visibility to all of the objects being passed as parameters and to the object providing the method.

Establishing communication using parameters works well for an object that is using a method that is provided by one component that uses some other component object as a parameter. Trying to use this option in situations other than where a container object uses methods of a component dictates that the programmer "pass through" objects to methods that need them as parameters. This is not

advisable because it causes (possibly unnecessary) coupling in the system. There are no examples of using objects as parameters to methods in the Flight Simulator.

The last option C++ offers to provide communication paths is through the use of static data members and methods. Using this option is limited to situations in which only one object will exist in the system. The advantages to using static objects are that they exist autonomously in the system and access to the methods of the static object are available without having to first instantiate an object of the class in which the static object is defined.

The Static Timer static object is an example of a static object in the Flight Simulator. There will only be one object that tracks the simulator time. The Flight Simulator object is responsible for keeping the time correctly while other objects simply access what the timing values are. No pointers to the Timer are passed down to lower level objects by the Flight Simulator nor is the Timer declared as an attribute of the Flight Simulator object. The Timer exists on its own and is easily accessed by everything.

E.3.2 Hiding Data The second part of providing object visibility pertains to specifying the exact visibility of attributes and methods of each object with respect to all others in the system. This is accomplished using the C++ keywords to define the visibility of an object: "private", "protected", "public", and "friend".

E.3.2.1 C++ Visibility Mechanisms The keyword "private" is used to designate information that will be visible only to an object of that class. The keyword is used to specify the visibility of inheritance relationships and to parts of the class itself. Anything that can be included in a class description can be hidden with the "private" keyword. The "protected" keyword operates just like the "private" keyword except with respect to derived classes, which can access any protected members as if they were "public". The "public" keyword indicates that an inheri-

tance relationship and/or information in the object is not hidden from users of the object.

The "private", "protected", "public" and "friend" keywords are sometimes confusing to use, especially when inheritance is thrown into the problem. The visibility of an object of a class is fairly straightforward when no inheritance is involved. The private information of the object cannot be used while everything declared as public is fair game. If a class is designated as a friend, then an object of the friend class can access anything in the object in question. Adding inheritance makes things a bit more complex.

Consider the case where a programmer wishes to use an object of a class that serves as a base class for some other derived class(es). All of the data and methods declared as "protected" in the base class are "private" as far as the user of the base class object is concerned. The fact that the class even serves as a base class is not visible to the user of the object.

Now consider the case where the programmer wishes to use an object of a derived class. The visibility to data and methods of the derived class is unaffected by the fact that the class was derived from another. Visibility to the methods and data of the base class through the object of the derived class is determined by whether the inheritance relationship was declared as "private" or "public".

If the inheritance relationship was declared as "private", then a user of an object of the derived class would not be able to access anything provided by the base class. Conversely, if the relationship was defined as "public", a user of an object of the derived class would have access to the public information offered by the base class. The protected and private information of the base class would still remain unaccessible to the user of an object of the derived class.

The fact that the inheritance was public or private has no effect upon what the derived class can access from the base class. The declaration of the inheritance only

affects the visibility that a user of an object of a derived class has with respect to the public information offered by the base class. The visibility that a derived class has with respect to it's base class is determined by the visibility defined in the base class.

The derived class would be able to access the information in the base class just as a user of an object of the base class would with one exception. All details in the base class that were declared as "protected" could be accessed as though they were "public" by the derived class. Given this, there is no reason to declare a derived class as a "friend" to the base class. Visibility should be granted through the use of the "protected" keyword.

Adding more classes to an inheritance tree complicates matters further. However, the previous rules still apply. The best thing to do is consider a class as an amalgamation of everything from which it was derived. The visibility rules concerning inheritance relationships are unaffected by further inheritance established in lower level derived classes.

For example, consider a class X that serves as a base class for class Y and the inheritance is hidden. Another class Z that was derived from Y would not be able to access anything in class X. The fact that the inheritance between Y and Z was public or private would have no effect upon the fact that the inheritance between X and Y was hidden. As far as Z is concerned, Y is not inherited from anything.

E.3.2.2 Hiding Inheritance Relationships Establishing the visibility of an inheritance relationship centers on whether the public methods of the base class should be accessible through the derived class. If an inheritance relationship is declared as public, then users of objects of the derived class can use the public methods offered by the base class as if they were provided by the derived class. If the inheritance relationship is declared as private, then a user of an object of the derived class can only access methods provided by the derived class. Access to

everything in the base class would be prohibited.

In general, inheritance relationships should be hidden only if all of the (now) hidden public methods of parent classes are somehow replaced by the derived class. One of the basic ideas behind inheritance is to be able to reuse existing classes in order to build a more specialized object. If the parent classes are hidden, then reusability is compromised. Wanting to hide methods of base classes could also be an indication that the relationship between the classes is incorrect (see section 4.3.5.1). Inheritance relationships should be declared as "public".

There is an exception to this rule. It may be necessary in certain circumstances to have a derived class override a method offered by a base class. For example, suppose derived class B inherits from base class A. If class A offered a particular method that B wanted to replace, then it may be necessary to hide the inheritance relationship in order to prevent a user of object B from calling the method that A offered.

The C++ compiler can determine which method to call given the class of the object and the parameter profile of the method. Given the class, the compiler will call the method offered by that class if the parameter profile of the method called and the method offered both match. In addition, the compiler will call the method offered by the class B even if the parameter profiles are the same in classes A and B. If no method offered by class B matches the requested method, then the compiler will start to look up the inheritance chain for a method that does match the name and parameter profile of the method requested. The first match found is the one called.

The only time that it would be required to hide the inheritance relationship is when class B provides a method to override class A's version and the two methods do not have the same parameters. If the inheritance was public and a user were to request a method that matches class A's parameter profile, then class A's version

would be called. The programmer may not want a potential user of the object to know that class A's version even exists.

In this instance, a programmer might want to hide the inheritance between class A and class B. If a user were to try and use class A's version of the method, the compiler would catch the attempt as a syntax error. The bad part of this is that it may force the programmer to replace methods that are hidden in class A with nothing more than one line methods in class B to call the counterpart method in class A. This particular situation never arose during the implementation of the flight simulator.

E.3.2.3 Hiding Information Within an Object All data members of a class should be hidden. Except in situations where a class will serve as a base class, all data members should be declared as "private". In situations where a class will serve as a base class, use the "protected" keyword to hide data members from all classes except the derived classes of the base class. No object should be allowed access to the data of another object outside of the methods offered to access the data. Allowing unlimited access to data members violates the principle of information hiding.

Given that all data members should be hidden, the real focus of hiding information is the visibility to the methods that an object may offer. The "private", "protected" and "public" keywords work the same for methods as they do for data members. In general, all methods defined in the analysis and high level design phases should be public. It was my experience that the majority of private methods were added during low level design.

Most of the private methods present in the Flight Simulator were added during implementation in order to implement higher level methods. They are used only by the class in which they are defined. They do not represent enough functionality in and of themselves to be useful. Thus, they were hidden to prevent their use.

These lower level methods should not be hidden from derived classes. They

make the base class more reusable in that they provide the designer of the derived classes with very cohesive "black boxes" from which to compose methods in the derived class. Given that reusing a base class is a "white box" exercise, these methods also provide an amount of information hiding in this situation.

While the C++ language offers a lot of control over the visibility of an object, there are times when even more power is required. For example, it is not possible with C++ to specify which specific classes can access specific methods of a class. In other words, there is no way to specify a list of classes that can use a particular method of another class. The closest capability that C++ offers to this very high degree of control is the "friend" keyword.

The "friend" keyword is used to specify which classes have full access to the class in which they are declared as a "friend". Unfortunately, this is an all or nothing proposition. The friend class has the capability to use all private data and methods. Consequently, this is a very powerful and potentially dangerous capability.

Under no circumstances should the friend class directly modify the private/protected data of another class. The only acceptable reason for designating a friend class is to allow the friend class access to methods that should not be available to other classes. In spite of this, the fact that a friend class had to be designated in the first place might indicate a high(er) degree of coupling between the two classes. The use of the "friend" keyword should be highly scrutinized.

E.4 Conclusion

This appendix detailed when and why to use specific C++ language constructs in implementing an object-oriented design. Deciding how to represent objects in the design is based upon the functionality offered by the candidate object. The object could be implemented as a data type, through the class construct and as a static object. The C++ class construct should be used most often.

Establishing paths between the objects in the design can be accomplished in five different ways: 1) inheritance, 2) composition, 3) Using a pointer to an object, 4) passing an object as a parameter to a method and 5) using static objects. Each strategy has it's own advantages and disadvantages. Using the proper one depends upon the situation.

Defining the visibility provided by an object can be controlled with the four C+ keywords "private", "public", "protected" and "friend". The C++ programmer can exercise fine grained control over the accessibility to the methods and data of an object and of a class. In general, all data members should be hidden. Access to the object should only take place through the methods of the class. The C++ keywords used to specify visibility allow the programmer to take full advantage of encapsulation necessary for implementing an object-oriented design.

Bibliography

1. Bailin, Sidney C. "An Object-Oriented Requirements Specification Method," *Communications of the ACM, 32:* 608-622 (May 1989).
2. Bézevin, Jean. "Some Experiments in Object-Oriented Simulation," *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1987*. 394-405. New York: published in ACM SIGPLAN Notes, volume 22, October 1987.
3. Booch, Grady. *Object-Oriented Design With Applications*. Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991.
4. —. "Object-Oriented Development," *IEEE Transactions on Software Engineering 12:* 211-221 (February 1986).
5. — and Michael Vilot, "The Design of the C++ Booch Components," *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA) 1990*. 1-11. New York:published in ACM SIGPLAN Notes vol 25, October 1990.
6. Boyd, Captain Andrew D. 1991. *A Formal Definition of the Object-Oriented Paradigm for Requirements Analysis*. MS thesis. AFIT/GSS/ENG/91D-3. School of Systems and Logistics, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
7. Brooks, Frederick P., Jr. "No Silver Bullet — Essence and Accidents of Software Engineering," *IEEE Computer 20:* 10-19 (April 1987).
8. Brunderman, John A. 1991. *Design and Application of an Object-Oriented Graphical Database Management System for Synthetic Environments*. MS thesis. AFIT/GA/ENG/91D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
9. Cargill, T. A. "Pi: A Case Study in Object-Oriented Programming," *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1986*. 350-360. New York: published in ACM SIGPLAN Notes, volume 21, October 1986.
10. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis, 2nd ed.* Englewood Cliffs, NJ: Yourdon Press, 1991.
11. —. *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press, 1991.
12. Cox, Brad J. *Object Oriented Programming — An Evolutionary Approach*. Menlo Park, CA: Addison-Wesley Publishing Company, 1987.

13. —. "Planning the Software Industrial Revolution," *IEEE Software, 7:* 25-33 (November 1990).

14. Duckett, Donald P. 1991. *The Application of Statistical Estimation Techniques to Terrain Modeling Here*. MS thesis. AFIT/GCE/ENG/91D-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

15. Eldredge, David L. and others. "Applying the Object-Oriented Paradigm to Discrete Event Simulations Using the C++ Language," *Simulation, 54:* 83-91 (February 1990).

16. Embley, David W. and Scott N. Woodfield. "Assessing the Quality of Abstract Data Types Written in Ada," *Proceedings of the 10th International Conference on Software Engineering*. 144-153. Washington DC: IEEE Computer Society Press, 1988.

17. Filer, Captain Robert E. *A 3-D Virtual Environment Display System*. MS thesis. AFIT/GCS/ENG/89D-2. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

18. Foley, James and others. *Computer Graphics, Principles and Practices (2nd ed)*. New York:Addison-Wesley Publishing Company, 1990.

19. Freeman-Benson, Bjorn N., and others, "An Incremental Constraint Solver," *Course Notes for course C-22 — Object and Constraint Paradigms for Graphics. Course was given during SIGGRAPH 1991 — 18th International Conference On Computer Graphics and Interactive Techniques*. VII-25 - VII-34. ACM SIGGRAPH, August 1991.

20. Gerken, Mark J. 1991. *An Event Driven State Based Interface for Synthetic Environments*. MS thesis. AFIT/GCS/ENG/91D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

21. Gorlen, Keith E., Sanford M. Orlow and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. New York: John Wiley and Sons, 1990.

22. Gossain, Sanjiv and Bruce Anderson. "An Iterative-Design Model for Reusable Object-Oriented Software," *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming: Systems, Languages and Applications (ECOOP/OOPSLA) 1990*. 12-27. New York:published in ACM SIGPLAN Notes vol 25, October 1990.

23. Halbert, David C, and Patrick D. O'Brien. "Using Types and Inheritance in Object-Oriented Programming," *IEEE Software, 6:* 71-79 (September 1987).

24. Haviland, Keith and Ben Salama. *UNIX System Programming*. New York: Addison-Wesley, 1987.

25. Henderson-Sellers, Brian and Julian M. Edwards. "Object-Oriented Systems Lifecycle," *Communications of the ACM, 33:* 142-159 (September 1990).

26. Johnson, Ralph E. and Brian Foote. "Designing Reusable Classes," *The Journal of Object-Oriented Programming, 1,2:* 22-35 (June/July 1988).

27. Jordan, David. "Implementation Benefits of C++ Language Mechanisms," *Communications of the ACM, 33:* 61-64 (September 1990).

28. Korson, Tim and John D. McGregor. "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM, 33:* 41-60 (September 1990).

29. Laffra, Chris. "Object-Oriented Methods for Graphics," *Course Notes for course C-22 — Object and Constraint Paradigms for Graphics. Course was given during SIGGRAPH 1991 — 18th International Conference On Computer Graphics and Interactive Techniques.* I-1 - I-33. ACM SIGGRAPH, August 1991.

30. Lee, Kenneth J. and others. *An OOD Paradigm for Flight Simulators, 2nd Edition.* Prepared for the Electronic Systems Division, technical report number ESD-TR-88-31 by the Software Engineering Instititute, September 1988. (AD-A204 849).

31. — and Michael S. Rissman. *An Object-Oriented Solution Example: A Flight Simulator Electrical System.* Prepared for the Electronic Systems Division, technical report number ESD-TR-89-5 by the Software Engineering Institute, February 1989. (AD-A219 190).

32. Lieberherr, Karl J. and Ian M. Holland. "Assuring Good Style for Object-Oriented Programs," *IEEE Software, 6:* 38-48 (September 1989).

33. — and others. "Object-Oriented Programming: An Objective Sense of Style," *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1988.* 323-334. New York: published in ACM SIGPLAN Notes, volume 23, October 1988.

34. Meyer, Bertrand. *Object-Oriented Software Construction.* New York: Prentice Hall, 1988.

35. —. "Reusability: The Case for Object-Oriented Design," *IEEE Software 4:* 50-63 (March 1987).

36. Miller, Katherine S. and others. *Object-Oriented Software Requirements Specification for the UH-1 Helicopter Flight Simulator.* Prepared for PM Trade, AMCPM-TND-ED, report number MDA 903-87-D-0056 by IIT Research Institute, June 1990. (AD-A225 041).

37. Mullin, Mark. *Object-Oriented Program Design With Examples in C++.* Menlo Park, CA: Addison-Wesley Publishing Company, 1989.

38. Olson, Robert A. 1991. *Techniques to Enhance the Visual Realism of a Synthetic Environment Flight Simulator. MS thesis.* AFIT/GCS/ENG/91D-16. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

39. Parnas, David L. and others. "Enhancing Reusability With Information Hiding," *IEEE Tutorial on Software Reusability*, Washington DC: IEEE Computer Society Press, 1984.

40. Platt, Philip A. 1990. *Real-Time Flight Simulation and the Head-Mounted Display — An Inexpensive Approach to Military Pilot Training.* MS thesis. AFIT/GCS/ENG/90D-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

41. Popken, Douglas A. *Object-Oriented Simulation Environment for Airbase Logistics: Interim Report, December 1989 - September 1990.* Air Force Human Resources Lab (AFHRL-TP-90-78), November 1990 (AD-A228 055).

42. Pressman, Roger S. *Software Engineering, A Practitioner's Approach.* New York: McGraw-Hill Book Company, 1982.

43. Rebo, Captain Robert K. *A Helmet-Mounted Virtual Environment Display System System.* MS thesis. AFIT/GCS/ENG/88D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

44. Rochkind, Marc J. *Advanced UNIX Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

45. Rolfe, J. M. *Flight Simulation.* New York: Cambridge University Press, 1986.

46. Rubin, Kenneth S. "Reuse in Software Engineering: An Object-Oriented Perspective," *Proceedings of COMPCON Spring 90: 35th IEEE Computer Society International Conference.* 340-346. Los Alamitos: IEEE Computer Society Press, 1990.

47. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

48. Sanderson, Peter D. and Lawrence L. Rose. "Object-Oriented Modeling Using C++," *Proceedings of the 21st Annual Simulation Symposium.* 143-156. Washington DC: IEEE Computer Society Press, 1988.

49. Schachter, Bruce J. *Computer Image Generation.* New York: John Wiley & Sons, Inc., 1983.

50. Seidewitz, Ed. "General Object-Oriented Software Development Background and Experience," *The Journal of Systems and Software, 9:* 95-108 (1989).

51. Shlaer, Sally and Stephen J. Mellor. "An Object-Oriented Approach to Domain Analysis," *ACM SIGSOFT Software Engineering Notes, 14:* 66-77 (July 1989).

52. Silicon Graphics, Incorporated. *Graphics Library Programming Guide*, version 2.0. Mountain View, CA, 1990.

53. Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1986*. 38-45. New York: published in ACM SIGPLAN Notes, volume 21, October 1986.

54. Somerville, Ian. *Software Engineering*. Menlo Park, CA: Addison-Wesley Publishing Company, 1989.

55. Stroustrup, Bjarne. *The C++ Programming Language*. Menlo Park, CA: Addison-Wesley Publishing Company, 1986.

56. Stroustrup, Bjarne. "What is Object-Oriented Programming?," *IEEE Software, 5:* 10-20 (May 1988).

57. Umphress, Major David A. 1990. Class handout distributed in CSCE 593, Systems and Software Analysis. School of Engineering. Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

58. Wasserman, Anthony I. and others. "The Object-Oriented Stuctured Design Notation for Software Design Representation," *IEEE Computer, 23:* 50-63 (March 1990)

59. Wegner, Peter. "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger, 1:* 7-87 (August 1990).

60. Winblad, Ann L. and others. *Object-Oriented Software*. Menlo Park, CA: Addison-Wesley Publishing Company, 1990.

61. Wirfs-Brock, Rebecca J. and Brian Wilkerson. "Object-Oriented Design: A Responsibility-Driven Approach," *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA) 1989*. 71-75. New York: published in ACM SIGPLAN Notes, volume 24, October 1989.

62. Young, O. M. *ADSIM, An Object-Oriented Hybrid Time-Stepped/Event-Based Large Scale Air Defense Simulation*. Contract F19628-89-C-0001. Bedford MA: The Mitre Corporation, July 1990 (AD-B147 124).

63. Zyda, Michael. "Flight Simulators for Under $100,000," *IEEE Computer Graphics and Applications, 8:* 19-27 (January 1988).